# Constant-RMR Implementations of CAS and Other Synchronization Primitives Using Read and Write Operations

Wojciech Golab*
HP Labs, Palo Alto, USA
wojciech.golab@hp.com

Vassos Hadzilacos*
University of Toronto, Canada
vassos@cs.toronto.edu

Danny Hendler
Ben-Gurion University, Israel
hendlerd@cs.bgu.ac.il

Philipp Woelfel*
University of Calgary, Canada
woelfel@cpsc.ucalgary.ca

May 28, 2011

**Abstract**

We consider asynchronous multiprocessors where processes communicate only by reading or writing shared memory. We show how to implement consensus, compare-and-swap (CAS) and other comparison primitives, as well as load-linked/store-conditional (LL/SC) using only a *constant* number of remote memory references (RMRs), in both the cache-coherent and the distributed-shared-memory models of such multiprocessors. Our implementations are blocking, rather than wait-free: they ensure progress provided all processes that invoke the implemented primitive are live.

Our results imply that *any* algorithm using read and write operations, comparison primitives, and load-linked/store-conditional, can be simulated by an algorithm that uses read and write operations only, with at most a constant-factor increase in RMR complexity.

**Keywords:** Comparison primitives, consensus, load-linked/store-conditional, mutual exclusion, remote memory references, shared memory.

*Supported partially by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

# 1 Introduction

Work on synchronization in shared memory multiprocessors has largely focused on the asynchronous model, either with or without crash failures. In wait-free synchronization, each process must make progress through its own steps regardless of the execution speeds or crash failures of others; in blocking synchronization, processes may busy-wait for others by repeatedly accessing shared memory, and so progress is guaranteed only when every active process is live. (A process is live if, whenever it begins executing an algorithm, it continues to take steps until the algorithm terminates.)

In this paper we focus on blocking synchronization in asynchronous multiprocessors where processes communicate through shared memory. A natural way to measure the time complexity of algorithms in such multiprocessors is to count the number of memory accesses. This measure is problematic for blocking algorithms because, in this case, a process may perform an unbounded number of memory accesses while busy-waiting for another process. (For example, this happens in a mutual exclusion algorithm when a process waits for another to clear the critical section.) Instead, we can measure the time complexity of an algorithm by counting only *remote memory references* (RMRs), i.e., memory accesses that traverse the processor-to-memory interconnect. *Local-spin* algorithms, which perform busy-waiting by repeatedly reading shared variables that can be accessed *locally*, achieve bounded RMR complexity and have practical performance benefits [4].

The classification of memory accesses into local and remote depends on the memory model of the multiprocessor: In the *distributed shared memory* (DSM) model, a variable's physical address determines locality with respect to a processor, each variable being local to exactly one processor and remote to all others. In the *cache-coherent* (CC) model, processors operate on cached copies of shared variables, and it is the state of a processor's local cache combined with the action of the *coherence protocol* (which keeps consistent copies of a variable in different caches) that determines locality. A memory access is local if it results in a cache hit and can be resolved without accessing main memory or a remote cache; a memory access is remote otherwise. To analyze the worst-case RMR complexity of an algorithm, we assume that each process runs on a distinct processor. (For this reason, we speak of processes and processors interchangeably.)

The main theme of this paper is that certain popular synchronization primitives can be implemented efficiently in software, at least in terms of RMRs, from simpler ones. A synchronization primitive, in this context, is a type of operation that acts on a shared state abstractly represented by a memory word. We model an implementation of a set $\mathcal{S}$ of primitives as a typed shared object that supports an atomic operation on its state corresponding to each primitive in $\mathcal{S}$. Such an object supporting only read and write primitives is called a *read/write register* (or just *register*).

**Summary of results.**

**(1)** All *comparison* primitives [2], a class of synchronization primitives that includes the popular *compare-and-swap* (CAS) primitive, can be implemented using read and write operations with only a constant number of RMRs, in both the DSM and CC models. The same holds for the *load-linked/store-conditional* (LL/SC) pair of primitives. In both cases we show how to make the implementation *readable* and *writable* (i.e., we show how to support read and write primitives on the shared object).

**(2)** Our constant-RMR implementations can be made *locally-accessible* just like their hardware-implemented counterparts. In the DSM model, this means that the implemented object behaves as if it is local to some processor, and so some designated process can access the object without performing any RMRs. (This is nontrivial because the implemented object may use internally many base objects, not all local to the designated process.) Similarly, in the CC model the object behaves as if it can be cached, meaning that certain operations on an "in-cache" object cost

no RMRs; whether an object is "in-cache" depends on the prior history of the execution and the coherence protocol. (This again is nontrivial because the implemented object may use internally many base objects and access them in complex ways.)

**(3)** As a consequence of (1) and (2), any CC or DSM shared memory algorithm using read, write, comparison primitives and LL/SC can be simulated by an algorithm that uses only read and write operations, with only a constant-factor increase in the RMR complexity, while preserving other important properties.

Our constant-RMR implementations of comparison primitives and LL/SC are obtained in a series of steps. We first show how to transform *any leader election* algorithm that uses read and write operations into a *name consensus* algorithm that uses read and write operations and has the same worst-case RMR complexity to within a constant factor. (In a leader election algorithm, exactly one active process declares itself the winner, and all others declare themselves losers. In a name consensus algorithm, all active processes agree on one of their IDs.) Since there is a constant-RMR leader election algorithm [13], this transformation yields a constant-RMR name consensus algorithm. This efficient name consensus algorithm is used, in turn, to obtain constant-RMR CAS and LL/SC implementations from reads and writes. Finally, we observe that using CAS and no additional RMRs, one can easily implement any combination of comparison primitives.

**Related work and implications of our results.** Herlihy has shown that synchronization primitives vary widely in their ability to support *wait-free* implementations, and can be classified in the *wait-free hierarchy*, where the level of a primitive corresponds to its power [16]. For example, CAS together with read and write operations supports wait-free implementations of arbitrary objects shared by *any* number of processes; as a result, CAS is at the top level of the wait-free hierarchy. In contrast, the primitive *fetch-and-store* (FAS), together with read and write operations, supports wait-free implementation of arbitrary objects shared by at most *two* processes; as a result, FAS is at level two of the wait-free hierarchy. (FAS atomically reads a shared memory location and overwrites it with a value that is fixed in advance.)

As regards *blocking* synchronization, however, all primitives can be implemented using only read and write operations, by using such operations to implement mutual exclusion [9]. Thus, it is not meaningful to compare the power of two primitives by asking whether one can be used, along with read and write operations, to implement the other. Instead of comparing the power of two primitives based on computability, it is natural to ask whether we can base such a comparison on complexity — specifically, the RMR complexity of implementing a given primitive using read and write operations. From this point of view, we define the RMR complexity of a primitive, denoted $\mathcal{C}$, as the minimum of worst-case RMR complexity over all implementations of that primitive using read and write operations; and we say that a primitive $\mathsf{S}$ is stronger than a primitive $\mathsf{W}$ if and only if $\mathcal{C}(\mathsf{S})$ is greater asymptotically than $\mathcal{C}(\mathsf{W})$.

Looking at the relative power of primitives from this perspective reveals a landscape very different from that of Herlihy's wait-free hierarchy [16]. Some primitives classified as strong in the wait-free hierarchy have low RMR-cost implementations from read and write operations, and are weak in their ability to solve mutual exclusion efficiently in terms of RMRs. Conversely, some primitives classified as weak in the wait-free hierarchy have inherently high RMR-cost implementations from reads and writes, and yield the most RMR-efficient mutual exclusion algorithms. For example, CAS is at the top of the wait-free hierarchy but, as we show in this paper, it can be implemented from read and write operations using only a constant number of RMRs. On the other hand, FAS is only at level two of the wait-free hierarchy, but any implementation of FAS from read and write operations requires $\Omega(\log N)$ RMRs in the worst case (where $N$ is a parameter of the implementation that denotes the number of processes that can access it). This follows from the

fact that mutual exclusion can be solved with $O(1)$ RMRs per passage through the critical section using FAS along with read and write operations [6], but requires $\Omega(\log N)$ RMRs per passage in the worst case using only reads and writes [5]. The same holds for the primitive *fetch-and-add* (FAA) [4], which is also weak in the wait-free hierarchy.

Anderson and Kim were the first to propose a way of ranking synchronization primitives according to their power for solving mutual exclusion efficiently (under the RMR complexity measure) [1], and to contrast this approach with Herlihy's wait-free hierarchy [16]. To that end, they defined for each primitive (and value of $N$) a numerical rank $r$ that captures in a particular way the primitive's ability to break symmetry among $N$ processes that apply it concurrently to the same variable. They then showed that a primitive of rank $r \geq 2$ can be used, in conjunction with read and write operations, to solve mutual exclusion using $O(\max(1, \log_r N))$ RMRs per passage for any $N$. It is not known whether the latter bound is tight in general, although it is tight for common primitives such as reads and writes, CAS, FAS, and FAA.

Our ranking of synchronization primitives is similar to Anderson and Kim's in that it captures, at least for common primitives, their power to efficiently solve mutual exclusion for any $N$. The advantage of our ranking scheme over Anderson and Kim's is that it is based on a simpler property of "strength", namely the RMR complexity of implementing a primitive using read and write operations. This property is easier to define than Anderson and Kim's rank. Moreover, the RMR complexity bound obtained to evaluate a primitive's strength, in the sense we propose, is of independent interest.

Our results also have an interesting implication regarding mutual exclusion. To explain this we first recall certain facts about the RMR complexity of mutual exclusion. The most RMR-efficient mutual exclusion algorithm known to date that uses only read and write operations is one devised by Yang and Anderson; it performs $O(\log N)$ RMRs per passage through the critical section [24]. Attiya, Hendler and Woelfel showed that this is optimal [5], building on a prior $\Omega(\log N)$ lower bound by Fan and Lynch for a related but different cost model [11]. The optimality result holds for algorithms that use reads and writes only, and tightens a prior $\Omega(\log N / \log \log N)$ lower bound on RMRs by Anderson and Kim [2] that holds for a broader class of algorithms: those using reads, writes, and comparison primitives. Anderson and Kim posed the question whether $\Theta(\log N)$ is the tight worst-case RMR complexity lower bound for the latter class of algorithms [2]. We answer this in the affirmative through our result (3), in combination with the $\Omega(\log N)$ lower bound on RMR complexity of algorithms that use reads and writes only [5]. (For *first-come-first-served* (FCFS) mutual exclusion, the $\Omega(\log N)$ lower bound holds a fortiori, but the upper bound of $O(\log N)$ does not, because the simulation referred to by our result (3) breaks the FCFS property. The tight bound for FCFS mutual exclusion is established in [8].)

### 1.1 Organization

We describe our model of computation in Section 2. Our implementations of CAS and LL/SC are then derived in Sections 3–8 using a layered approach, which we explain in more detail below. Section 9 then describes how to implement comparison primitives in general, and states our main result (3). In Section 10 we conclude the paper by discussing open problems. To streamline the paper, we defer portions of technical analysis to Appendix A.

Presenting our implementations is challenging as we consider two very different shared memory architectures (CC and DSM) and two shared object types (CAS and LL/SC) that are not easily derived from each other in a manner that preserves correctness properties of interest in this paper. Furthermore, some of the algorithms are quite complex because the implementations are based upon weak base objects (i.e., atomic read/write registers). To simplify matters, we have chosen to

break down the construction into multiple layers, as illustrated in Figure 1.

The starting point in our construction is the leader election (LE) problem where, informally speaking, one of the participating processes is declared as the "winner" and all the others as "losers". A constant-RMR implementation of LE using only reads and writes was presented in [13]. In Section 3, we use LE to solve the Name Consensus (NC) problem where, informally speaking, participating processes agree on the name of the "winner". The constant-RMR reduction of NC to LE using only reads and writes is straightforward in the CC model, but considerably more complex in the DSM model. In Section 4 we use NC to construct a new primitive we call a pseudo-lock, similar to "one-shot" mutual exclusion. Pseudo-locks provide convenient synchronization machinery for dealing with concurrent operations in our other implementations. In Section 5, we use pseudo-locks to implement a block manager—an object similar in spirit to CAS, but specialized for recording pointers to special data structures we call blocks. Blocks can be used to record arbitrary state, such as the value of a CAS object.

In the layers above the block manager, we present a series of implementations with increasingly powerful properties. In order to address both CAS and LL/SC in a clean way, we introduce a new type called ECAS that generalizes these two primitives. Section 6 defines ECAS and presents a straightforward implementation of ECAS from a block manager. This implementation is sufficient for proving our main result (1) but not our main result (3), which motivates our definition of more powerful locally-accessible implementations in Section 7. Because the latter implementations are defined very differently in the CC and DSM architectures, we are forced from this layer onward to present a separate implementation for each architecture. Next, in Section 8 we show how to make our implementations writable—a feature omitted so far but obtainable easily using techniques developed in earlier sections. Finally, in Section 9 we explain how to instantiate our ECAS implementations to CAS and LL/SC objects that are strong enough to establish our main result (3). Comparison primitive in general are derived trivially from these CAS objects.

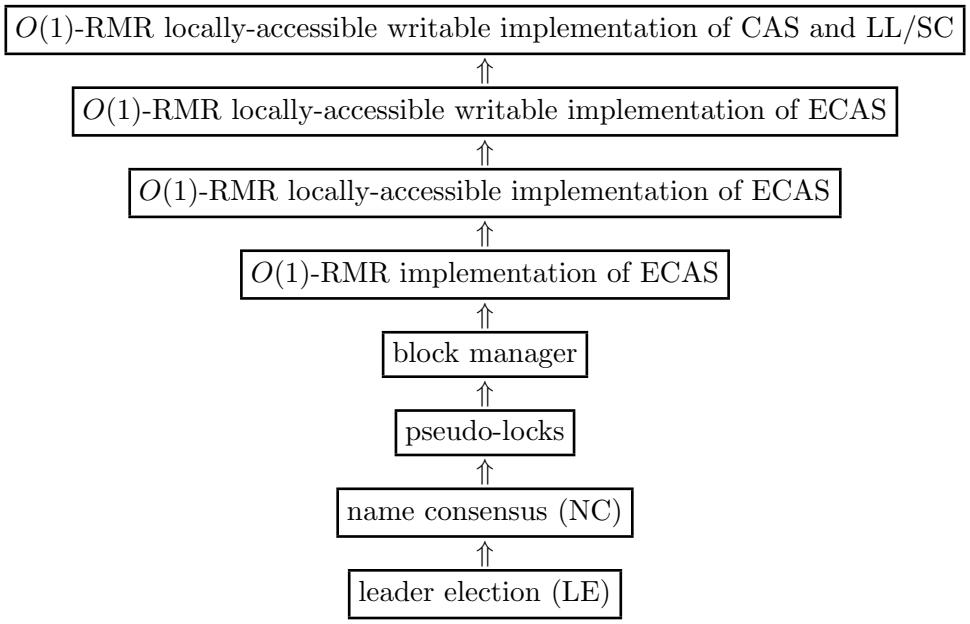


Figure 1: Implementation layers.

## 2 Model of Computation and Definitions

Our model of computation is based on Herlihy and Wing's [17].

**Processes and objects.** There are $N$ asynchronous *processes*. Processes do not fail. The set of processes is denoted $\mathcal{P} = \{p_1, p_2, ..., p_N\}$, and we say that $p_i$ has *ID* $i$. Processes communicate by applying operations on shared objects and receiving corresponding responses. Each process repeatedly applies such operations (one at a time) until it *terminates*, meaning that it has reached a special state where it remains indefinitely and takes no further action. A shared object represents a data structure with a well-defined set of states, as well as a set of *operation types*. The operation type determines the state transition that occurs when an operation of that type is applied to the shared object in a given state, as well as the response of the operation. It encodes the "signature" of the operation (including any arguments), as well as the ID of the process applying the operation. Processes and objects can be formally modeled as input/output automata [21], but here we adopt a more informal approach by describing the possible behaviours of processes and shared objects through pseudo-code. In pseudo-code, we refer to shared objects as *variables*.

**Steps.** A process interacts with shared objects by applying operations on these objects. We consider two types of operations: atomic and non-atomic. Atomic operations are instantaneous, and are represented as *atomic steps*. An atomic step where process $p$ applies an operation of type $ot$ to object $v$ and receives response $ret$ is denoted $(p, v, ot, ret)$. Non-atomic operations are represented using separate *invocation* and *response steps*. An invocation step where process $p$ invokes operation type $ot$ on object $v$ is denoted $(\text{INV}, p, v, ot)$. A response step where process $p$ receives response $ret$ from an operation execution on object $v$ is denoted $(\text{RES}, p, v, ret)$. We say that a response step *matches* an invocation step if the two steps are applied by the same process to the same shared object.

**Histories.** A *history* $H$ is a sequence of steps generated by processes. We explain how histories are generated later on and focus for now only on their building blocks. An *operation execution* in $H$ is a pair consisting of an invocation step and the next matching response step, or just an invocation step if no matching response follows. We call an operation execution *complete* in the former case, and *pending* in the latter. Operation execution $ox$ *precedes* operation execution $ox'$ in $H$ if the response of $ox$ occurs before the invocation of $ox'$ in $H$. We say that $ox$ and $ox'$ are *concurrent* in $H$ if neither precedes the other. A history $H$ is *sequential* if it only contains atomic steps, or if it only contains complete operation executions no two of which are concurrent. If $H$ is sequential, then $|H|$ denotes the number of atomic steps or operation executions in $H$.

For any history $H$ and set $P$ of processes, we denote by $H|P$ the subsequence of $H$ consisting of all steps by processes in $P$. Similarly, for any set $V$ of shared objects, we denote by $H|V$ the subsequence of $H$ consisting of all steps on objects in $V$. For a process $p$ or object $v$, we use $H|p$ and $H|v$ as shorthands for $H|\{p\}$ and $H|\{v\}$, respectively. We say that $H$ is a history *over* $V$ if $H = H|V$.

For any history $H$, we say that a process $p$ is *active* in $H$ if $H|p$ is not empty. An infinite history $H$ is *fair* if every process that is active in $H$ either takes infinitely many steps or terminates.

**Object types and conformity to a type.** Every shared object has a *type* $\tau = (\mathcal{S}, s_{init}, \mathcal{O}, \mathcal{R}, \delta)$ where $\mathcal{S}$ is a set of states, $s_{init} \in \mathcal{S}$ is the initial state, $\mathcal{O}$ is a set of operation types, $\mathcal{R}$ is the set of responses, and $\delta : \mathcal{S} \times \mathcal{O} \to \mathcal{S} \times \mathcal{R}$ is a (one-to-many) state transition mapping. The transition

mapping $\delta$ is intended to capture the behaviour of objects of type $\tau$, in the absence of concurrency, as follows: if a process applies an operation of type $ot$ to an object of type $\tau$ that is in state $s$, then the object may return to the process a response $r$ and change its state to $s'$ if and only if $(s', r) \in \delta(s, ot)$. An object $v$ *conforms* to type $\tau$ in a sequential history $H$ if the steps in $H|v$ are consistent with some sequence of transitions of $\delta$ starting from state $s_{init}$, in the following sense: Letting $ot_i$ and $ret_i$ denote the operation type and response of the $i$'th atomic step or operation execution in $H|v$, and letting $k = |H|v|$, there exists a sequence $\langle s_0, s_1, s_2, ..., s_k \rangle$ of states (in $\mathcal{S}$) such that $s_0 = s_{init}$, and $(s_i, ret_i) \in \delta(s_{i-1}, ot_i)$ holds for all $i \leq k$.

**States.** Let $H$ be a sequential history over some set $\mathcal{B}$ of objects, such that every object $v \in \mathcal{B}$ conforms to its type in $H$. For any $k \leq |H|$, the *state of the system* (or simply "the state") after $k$ atomic steps or operation executions in $H$ is denoted $H[k]$, and consists of the following: the state of each shared object and the *private state* of each process. The private state of a process comprises the values of private variables, in particular a "program counter" that determines the next pseudo-code statement executed by that process (and whether the process has terminated).

**Concurrent systems.** A *concurrent system* models algorithms where processes apply atomic steps on a set of shared objects. Formally, it is a tuple $S = (\mathcal{P}, \mathcal{B}, \mathcal{H})$ where $\mathcal{P}$ is the set of processes, $\mathcal{B}$ is the set of shared objects, and $\mathcal{H}$ is the set of histories of the concurrent system. We will define $\mathcal{H}$ informally through pseudo-code, consisting of one or more functions for each process, which are typically called according to some specific rules (e.g., each function is called at most once). The set $\mathcal{H}$ then consists of histories generated by recording an atomic step for each access to a shared object incurred by processes as they execute their functions.

**Implementations of shared objects.** An *implementation* describes how to simulate a *target object* of a particular *target type* using a set of *base objects* of specified types. It is formally denoted as a tuple $I = (\tau, \mathcal{P}, \mathcal{B}, \mathcal{H})$ where $\tau$ is the target object type, $\mathcal{P}$ is the set of processes, $\mathcal{B}$ is the set of base objects, and $\mathcal{H}$ is the set of histories. The histories in $\mathcal{H}$ are over the base objects in $\mathcal{B}$ and the target object of type $\tau$, denoted $O_\tau$. Informally, we describe an implementation using pseudo-code to define an *access procedure* for each operation type $ot$ of the target type and each process $p$. The pseudo-code for this access procedure describes how process $p$ applies an operation of type $ot$ to the target object, and computes the response of that operation, by applying operations on the base objects. The set $\mathcal{H}$ consists of histories generated by recording an invocation or response step on the target object whenever a process begins or finishes executing an access procedure (respectively), and an atomic step for each access to a base object in $\mathcal{B}$.

Two correctness properties are required in every implementation: linearizability (safety) and termination (liveness).

**Linearizability and termination.**

Linearizability [17] is widely accepted as a correctness condition for histories of shared object implementations. Informally, it states that each operation execution on the target object appears to take effect instantaneously at some point between the operation execution's invocation and response (or possibly not at all if the operation execution is pending). Formally, we first define for any history $H$ of an implementation $I = (\tau, \mathcal{P}, \mathcal{B}, \mathcal{H})$ a *completion*, which is a history $H'$ of invocation and response steps on the target object $O_\tau$ such that for every process $p$, $H'|O_\tau|p$ contains the same steps as $H|O_\tau|p$, except that for any operation execution $Op$ that is pending in $H$, either $Op$ is discarded from $H'$ or a matching response step follows the invocation step of $Op$

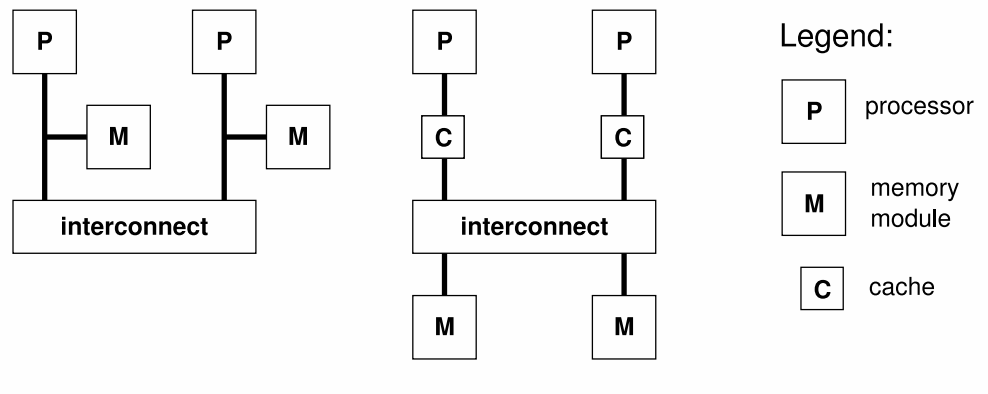


Figure 2: Shared memory architectures: DSM (left) and CC (right).

in $H'$. A history $H$ of an implementation $I = (\tau, \mathcal{P}, \mathcal{B}, \mathcal{H})$ is *linearizable* with respect to type $\tau$ if there exists a history $\bar{H}$ that satisfies the following properties:

(a) $\bar{H}$ is a sequential completion of $H$.
(b) The total order of operation executions in $\bar{H}$ is consistent with the partial order of the corresponding operation executions in $H$.
(c) The target object $O_\tau$ conforms to type $\tau$ in $\bar{H}$.

The *termination* property for a shared object implementation $I = (\tau, \mathcal{P}, \mathcal{B}, \mathcal{H})$ states that for any history $H$ of $I$, if $H$ is fair then every operation execution on the target object $O_\tau$ in $H$ is complete.

**Local and Remote Memory References.** In this paper, we consider the cache-coherent (CC) and distributed shared memory (DSM) multiprocessor architectures, which are illustrated in Figure 2.

In each architecture, each memory access is either *local* or *remote*, as discussed briefly in Section 1. We now describe these concepts in detail in the context of accesses to the most fundamental of all shared objects: atomic read/write registers. In the DSM model, locality is defined statically: each object is local to exactly one process and is remote to all others, and so counting RMRs is straightforward. In the CC model, however, whether an access to an object is local or remote depends on the type of coherence protocol, and the prior accesses to that object in the history under consideration. We consider two families of cache coherence protocols in this paper: *write-through* and *write-back* [23].

In a write-through protocol, to read an object $v$ a process $p$ must have a (valid) cached copy of $v$. If it does, $p$ reads that copy without causing an RMR; otherwise, $p$ causes an RMR that creates a cached copy of $v$. To write $v$, $p$ causes an RMR that invalidates (i.e., effectively deletes) all other cached copies of $v$, and writes $v$ to memory (in the same RMR). (The write-through protocol comes in two flavours: with cache invalidation and with cache update upon write. In this paper we consider only the invalidation version, as it is far more common in practice.)

In a write-back protocol, each cached copy is held in either "shared" or "exclusive" mode. To read an object $v$, a process $p$ must hold a cached copy of $v$ in either mode. If it does, $p$ reads that copy without causing an RMR. Otherwise, $p$ causes an RMR that: (a) eliminates any copy of $v$ held in exclusive mode, typically by downgrading the status of such a copy to shared and, if the exclusive copy was modified, writing that copy to memory; and (b) creates a local cached copy of $v$ held in shared mode. To write $v$, $p$ must have a cached copy of $v$ held in exclusive mode. If it

does, $p$ writes that copy without causing RMRs. Otherwise, $p$ causes an RMR that: (a) invalidates all other cached copies of $v$ and writes any modified copy held in exclusive mode back to memory; and (b) creates a cached copy of $v$ held in exclusive mode.

In both protocols, a read of object $v$ by process $p$ causes an RMR if and only if $p$ has no (valid) cached copy of $v$. The protocols differ in the RMRs caused by writes: in write-through, every write causes an RMR; in write-back, a write of $v$ by $p$ causes an RMR if and only if $p$ does not hold a local cached copy of $v$ in exclusive mode.

It is possible to define RMRs precisely in the CC model under certain assumptions that capture "ideal" cache behaviour (e.g., ignoring RMRs due to finite cache size and false sharing). For our purposes, however, it suffices to define simple rules by which we can bound from above the number of RMRs incurred in a history. (We state these for sequential histories over atomic read/write registers here, and then generalize to other types of shared objects in Section 7.) For any history $H$ of atomic steps over a read/write register object $v$, and for any process $p$, each atomic step by $p$ causes an RMR, except in the situations described below.

In the write-through CC model:

**(R)** *If $H'$ is a contiguous subsequence of $H$ where each atomic step is a read, then $p$'s atomic steps in $H'$ cause at most one RMR in total (to load $v$ into $p$'s cache).*

In the write-back CC model, condition (R) holds, and furthermore:

**(W)** *If $H'$ is a contiguous subsequence of $H$ where each atomic step is applied by $p$, then the atomic steps in $H'$ cause at most two RMRs in total (to load $v$ into $p$'s cache, and then possibly to promote $p$'s local copy of $v$ from shared to exclusive.)*

**Notation.** We use the following notational conventions. In pseudo-code, $p_i$ denotes the process ID $i$, and PID denotes the ID of the executing process. We denote by **read(***var***)** a read of shared variable *var*, returning the value read. Similarly, we denote by **write** *var* := *val* a write of *val* to shared variable *var*. We denote by **await** *cond* a busy-wait loop that repeatedly evaluates condition *cond*, and terminates when *cond* evaluates to true. The symbol $\triangleright$ in pseudo-code denotes access to a data structure field through a pointer, and is analogous to the operator –> in C++ (e.g., $d \triangleright f$ denotes a field $f$ in a data structure pointed to by $d$). We use a variety of typefaces in pseudo-code to distinguish various programming constructs: **reserved keyword**, *variable*, `FunctionName` and constantName. Comments are formatted in C++ style. Since we use pseudo-code to define both the transition mappping of a shared object type and a specific implementation of the type, we distinguish between the two by enclosing transition mapppings in a box.

# 3 Consensus

In this section, we obtain an $O(1)$-RMR consensus algorithm for $N$ processes using reads and writes only. We consider the special case of consensus known as *name consensus*, from which ordinary consensus follows by a straightforward transformation that preserves RMR complexity, with only $O(1)$ additional RMRs per process.

Roughly speaking, in the name consensus (NC) problem the active processes must all agree on a common value, which is the name (ID) of one of them. A process *wins* if its name is agreed upon and *loses* otherwise. The problem is formally defined as follows. First, since NC is a "one-shot" problem, processes must satisfy the following:

**Condition 3.1.** *Each process calls* `NameDecide()` *at most once.*

The correctness properties of name consensus are then defined as follows:

**Specification 3.2** (safety)**.** *For any history where Condition 3.1 holds:*

*(a) Each call to* `NameDecide()` *that terminates returns the ID of a process that has made a call to* `NameDecide()`*.*
*(b) No two calls to* `NameDecide()` *return different values.*

**Specification 3.3** (liveness)**.** *For any fair history where Condition 3.1 holds, each call to* `NameDecide()` *terminates.*

Name consensus is nontrivial in our model because the winner must be an active process, and not every process is required to be active; this rules out the naive algorithm that simply returns the ID of some fixed process.

We distinguish name consensus from the leader election (LE) problem, which was solved with $O(1)$ RMRs using reads and writes in [13]. In the leader election problem, each active process executes a function `LeaderElect()`, which returns win to exactly one process (the leader or winner), and lose to all others. More formally, leader election is specified as follows:

**Condition 3.4.** *Each process calls* `LeaderElect()` *at most once.*

**Specification 3.5** (safety)**.** *For any history where Condition 3.4 holds:*

*(a) If a call to* `LeaderElect()` *terminates, then it returns either* win *or* lose*.*
*(b) At most one call to* `LeaderElect()` *returns* win*.*
*(c) If each call made to* `LeaderElect()` *terminates, then exactly one such call returns* win*.*

**Specification 3.6** (liveness)**.** *For any fair history where Condition 3.4 holds, each call to* `LeaderElect()` *terminates.*

Leader election is trivial to solve using name consensus (by comparing the winner's ID to the caller's ID). Similarly, in the CC model name consensus is easy to solve using leader election with only $O(1)$ additional RMRs per process. An algorithm that does this is presented in Figure 3. In this algorithm, processes first execute a leader election algorithm $L$ (line **1**) and then either read or write a shared variable *leader* initialized to $\perp$. The winner of $L$ writes its ID to *leader* (line **2**) and returns its own ID; other processes wait for $leader \neq \perp$ (line **4**) and then return the value written to *leader* by the winner. We will refer to the corresponding concurrent system (see Section 2) as $\mathcal{A}_{NC\text{-}CC}$. It is straightforward to show that $\mathcal{A}_{NC\text{-}CC}$ satisfies Specifications 3.2 and 3.3, and that `NameDecide()` incurs only one more RMR than $L$ in the CC model. Thus, if we instantiate $L$ with

a LE algorithm that uses only reads/writes and $O(1)$ RMRs per process in the CC model, such as the algorithm given in [13], we obtain a NC algorithm that uses only reads/writes and $O(1)$ RMRs per process in the CC model.

**Declarations**

**Shared variables**:

*leader* – register, stores process ID or $\perp$, initially $\perp$

**Subroutines**:

$L$ – leader election algorithm

**Function** `NameDecide()`

**Output**: PID of winner

```
1 if L.LeaderElect() = win then
2 |   write leader := PID
3 else
4 |   await leader ≠ ⊥
5 end
6 return read(leader)
```

Figure 3: Name consensus algorithm for the CC model.

In the DSM model, the above algorithm is correct but has poor worst-case RMR complexity. This is because the variable *leader* is local to exactly one process, and for all others the busy-wait loop at line **4** may generate an unbounded number of RMRs. (At line **4** a process reads *leader* repeatedly until $leader \neq \perp$ holds, which may require an unbounded number of reads due to asynchrony.) In modifying this algorithm to achieve bounded RMR complexity in the DSM model, we must allow each process that does not win $L$ to busy-wait on its own locally-accessible shared variable until the winner is chosen. The winner must ensure that all of these variables are written, but it cannot write them itself using only $O(1)$ RMRs because there may be up to $N-1$ such variables, all remote to the winner. The technique presented in [13] for sharing work does not solve this problem because the winner does not know the IDs of all the processes that may be waiting for it (and hence are capable of sharing work). Nevertheless, a name consensus algorithm that builds on leader election is quite natural, and, as we show in the remainder of this section, is possible to construct with $O(1)$-RMR overhead in the DSM model using read and write operations. For the remainder of this section, we focus on the DSM model.

### 3.1 Name Consensus in the DSM Model: A High-Level Description

We now show how to solve name consensus at a cost of $O(1)$ RMRs per process in the worst case in the DSM model. Our implementation uses as a building block an $O(1)$-RMR leader election algorithm $L$ that uses only read/write registers, such as the one presented in [13], as well as some additional read/write registers. Our approach can also be used to construct a name consensus algorithm using *any* leader election algorithm, with at most a constant-factor increase in worst-case RMR complexity.

The way we use $L$ is derived from the following simple observation: *After any history of a leader election algorithm in which all active processes terminate, there is a "data flow" from the process elected leader to any other active process.* We can define the notion of data flow more precisely using graph-theoretic concepts. For each process $p$, we first define the following sets in the context of $p$'s execution of $L$:

- $W_p$ – set of processes to which the variables written remotely by $p$ are local
- $R_p$ – set of processes that wrote the values $p$ read remotely

  (N.B.: $R_p$ is not necessarily the set of processes to which the variables read remotely by $p$ are local. For example, if process $p$ reads a remote variable $v$ that is local to process $q$, and the value that $p$ reads in $v$ was written by process $r$, then $R_p$ records $r$ and not $q$.)

Next, let $G = (V, E)$ be the directed graph where $V$ is the set of processes, and $(p, q) \in E$ if and only if $q \in W_p$ (i.e., $p$ wrote remotely to $q$'s memory) or $p \in R_q$ (i.e., $q$ read remotely a value written by $p$). If $l$ is the process elected leader in $L$, then a data flow from $l$ to another process $p$ is a path from $l$ to $p$ in $G$. We will prove later (see Lemma 3.14) that there is such a path from $l$ to every active process. Intuitively, this is because if such a data flow does not exist, we can construct a new execution of algorithm $L$ in which only processes to which such paths do *not* exist are active, and they behave exactly as in the execution of $L$ that gave rise to $G$; these processes would elect $l$, which is not active in this new execution.

The high-level idea behind the NC algorithm is as follows. The leader $l$ elected in $L$ writes its ID in a variable *leader*, and will be the winner agreed on in NC. It then signals its out-neighbours in $G$ to let them know that the winner's name has been decided; once signalled, each neighbour repeats this step: it signals *its* out-neighbours; and so on. By our earlier observation, that there is a path from $l$ to *all* active processes in $G$, eventually all active processes will in fact be signalled, and each can simply read the winner's name from *leader*.

A number of issues must be addressed for this idea to work, and moreover to work with the required $O(1)$-RMR complexity per process:

(a) A process $p$ does not always know all its out-neighbours: it knows the out-neighbours in $W_p$, but does not necessarily know every $q$ such that $p \in R_q$.

(b) In fact, because of asynchrony, $p$ might not be able to *ever* discover some of its out-neighbours. For example, suppose that $p$ executes the NC algorithm (and within it $L$) and writes some register. After $p$ terminates, $q$ "wakes up" and reads a value written by $p$ while executing $L$, so that $p \in R_q$. Thus, $q$ is an out-neighbour of $p$, and yet $p$ has finished the NC algorithm and cannot be expected to signal $q$.

(c) Even if a process $p$ knows all its out-neighbours, it cannot signal them by simply writing into their local memory (while they busy-wait) because there may be many processes $q$ such that $p \in R_q$. For example, suppose that in the LE portion of the NC algorithm, $p$ writes a value that is read by all other processes. Then $p$ has $N - 1$ out-neighbours, and so if $p$ had to write a variable local to each, it would be using $\Theta(N)$ RMRs instead of $O(1)$.

We address these issues as follows. To solve (a) and (b), we use an idea introduced in [13]: a "handshaking protocol" that allows a process $p$ to synchronize with each out-neighbour $q$ such that $p \in R_q$ by either discovering $q$'s ID, or letting $q$ know (in case $q$ becomes active much later than $p$) that $p$ is "out of the picture". In the former case, $q$ waits for a signal from $p$. In the latter case, $q$ knows not to wait for $p$, and can read the winner's ID from some shared variable (e.g., one written by $l$ before $p$ executes its side of the handshaking protocol). Thus, $p$ need not know all of its out-neighbours (which solves (a)), and "latecomers" can discover the winner's ID easily (which solves (b)). Finally, for problem (c) we use a work sharing mechanism from [13] that spreads RMRs among the processes being signalled, namely the out-neighbours of $p$.

We now describe the building blocks of the name consensus algorithm in detail in Sections 3.2–3.4, and then present the NC algorithm itself in Section 3.5.

### 3.2 Instrumented Leader Election Algorithm

In order to compute the sets $R$ and $W$ defined earlier, processes execute an "instrumented" version of $L$ (denoted $\hat{L}$) rather than using $L$ directly. This algorithm returns the same response as $L$, and also computes $R_p$ and $W_p$ for every active process $p$. More precisely, we construct $\hat{L}$ from $L$ as follows: For every register $r$ initialized to value $x$ by $L$, initialize $r$ to $(\perp, x)$. For every uninitialized register $r$ that may be accessed in $L$, initialize $r$ to $(\perp, \tilde{x})$ for some arbitrary value $\tilde{x}$. Each active

process $p \in \mathcal{P}$ records $R_p, W_p \subseteq \mathcal{P}$ as private variables, both initialized to the empty set. To execute $\hat{L}$.`LeaderElect()`, a process $p$ begins by simulating its operations in $L$.`LeaderElect()`. If its next operation in $L$.`LeaderElect()` writes value $x$ to register $r$ that is local to $q$, then $p$ writes $(p, x)$ to $r$, and adds $q$ to $W_p$ if $q \neq p$. Process $p$ also simulates the change in private state following its write of $r$. If its next operation in $L$.`LeaderElect()` reads register $r$ that is local to process $q$, then $p$ reads $r$. If $(z, x)$ is the pair that $p$ read, then $p$ adds $z$ to $R_p$ if $z \neq \perp$, $z \neq p$, and $q \neq p$. Here $p$ also simulates the change in private state following its read of $r$, treating $x$ as the value read. Process $p$ repeatedly simulates its steps in $L$.`LeaderElect()` in this manner until termination, at which time $p$'s execution of $\hat{L}$.`LeaderElect()` produces the sets $R_p$ and $W_p$, and finally returns to $p$ the response of $L$.`LeaderElect()`.

They key properties of $\hat{L}$ are captured in the following lemma. Let $\mathcal{A}_{LE\text{-}I\text{-}DSM}$ denote the corresponding concurrent system.

**Lemma 3.7.** *For any history $H$ of $\mathcal{A}_{LE\text{-}I\text{-}DSM}$ where Condition 3.4 holds:*

*(a) Specifications 3.5 and 3.6 hold.*
*(b) Each call to $\hat{L}$.`LeaderElect()` incurs $O(1)$ RMRs in the DSM model.*
*(c) For each process $p$, the sets $R_p$ and $W_p$ generated during a call to $\hat{L}$.`LeaderElect()` have size $O(1)$.*

*Proof.* Let $L$ denote the $O(1)$-RMR leader election algorithm used to construct $\hat{L}$, and note that by our construction of $\hat{L}$, Condition 3.4 holds with respect to $\hat{L}$ in $H$. Furthermore, each process either receives the same response from $\hat{L}$.`LeaderElect()` as from $L$.`LeaderElect()` in $H$, or it does not complete its call to $\hat{L}$.`LeaderElect()`. Thus, if $H$ violates Specification 3.5 with respect to $\hat{L}$, then it does the same with respect to $L$. Similarly, if $H$ is fair and violates Specification 3.6 with respect to $\hat{L}$, then there must be a non-terminating call to $L$.`LeaderElect()` in $H$, and so $H$ violates Specification 3.6 with respect to $L$. Since $L$ satisfies Specifications 3.5 and 3.6, this implies $\hat{L}$ does also, and so (a) holds.

For parts (b)–(c), recall that $\hat{L}$.`LeaderElect()` simulates steps of $L$.`LeaderElect()`, performing an RMR at each step only if $L$.`LeaderElect()` does so, and adding an element to either $R$ or $W$ only when an RMR occurs. Since $R$ and $W$ are private variables, part (b) follows from the $O(1)$ RMR complexity of $L$. Similarly, for any process $p$ since $R_p$ and $W_p$ are initially empty, $|R_p| + |W_p|$ is bounded from above by the RMR complexity of $L$, which implies part (c). □

### 3.3 Handshaking Protocol

Our handshaking protocol is used by a process $p$ to synchronize with each out-neighbour $q$ such that $p \in R_q$. The protocol is similar in spirit to the one presented in [13], but somewhat simpler. For handshaking between $p$ and an out-neighbour $q \neq p$, the protocol relies on a two-process $O(1)$-RMR leader election algorithm for $p$ and $q$. Such an algorithm must be accessed according to the following etiquette (in addition to Condition 3.4):

**Condition 3.8.** *A two-process algorithm for processes $p$ and $q$ can only be accessed by $p$ and $q$.*

The particular LE algorithm we use is *local to $p$* meaning that, in addition to Specifications 3.5 and 3.6, it satisfies the following:

**Specification 3.9.** *A call to* `LeaderElect()` *by process $p$ incurs zero RMRs.*

An $O(1)$-RMR two-process LE algorithm satisfying this property is presented in detail in [13]. We omit the details for lack of space.

The LE algorithm is used for handshaking between $p$ and $q$ as follows: Process $p$ initiates an instance of the algorithm local to itself with each process $q \neq p$. Note that by Specification 3.9, $p$ incurs no RMRs in any of these. With respect to any particular process $q \neq p$, there are two outcomes of the LE algorithm for $p$ and $q$: If process $q$ wins, then we say that $q$ *contacted* $p$, otherwise $q$ *failed to contact* $p$. (This is similar to terminology used in [13].) In the former case ($q$ wins), $p$ eventually loses, and knows that $q$ is an active out-neighbour of $p$ such that $p \in R_q$. Thus, $q$ can wait for a signal from $p$, and $p$ knows that it must signal $q$. In the latter case ($q$ loses), $p$ eventually wins, and behaves as if $q$ were not active. Thus, $p$ does not signal $q$, and hence $q$ does not wait for a signal from $p$.

Note that up to $N-1$ processes may contact $p$, and that $p$ incurs zero RMRs handshaking with these processes, since the LE algorithm used is local to $p$. Also, note that $p$ may contact $q$ even if $q$ failed to contact $p$, since there may be two "sessions" of the handshaking protocol between $p$ and $q$, running in "opposite directions."

### 3.4 Signalling Mechanism

As mentioned before, our name consensus algorithm disseminates the leader's ID across the data flow graph $G$. The straightforward algorithm for doing this, whereby each process $p$ signals all its out-neighbours in $G$ (that it is aware of) by writing into their local memory, is too expensive in terms of RMRs. Instead, we use a signalling mechanism that shares the workload among $p$'s neighbours.

Informally, the signalling mechanism works as follows. When $p$ needs to communicate the leader's ID to a subset $\mathcal{N}_p$ of its neighbours (e.g., those which $p$ has contacted), it builds a chain of the IDs from $\mathcal{N}_p$ *in its local memory*. Process $p$ then signals the first process in the chain, say $q$, by writing $p$'s ID in a designated location in $q$'s memory. This costs $p$ a single RMR. Each process $q$ that is signalled in this way then reads the leader's ID from $p$'s memory and signals the next process in the chain (if any), whose ID it also reads from $p$'s local memory. The handshaking protocols executed prior to this ensure that for each such process $p$, all the processes in $\mathcal{N}_p$ wait for $p$'s (either direct or indirect) signal, as otherwise the signalling mechanism breaks.

The signalling mechanism consists of subroutines `signal`, `wait`, and `wait-any`, which are presented in Figure 4. Function `signal(`$P$`)` tells the processes in set $P$ that some event, for which they are waiting, has occurred. Function `wait(`$q$`)` blocks until a signal by process $q$ occurs. Function `wait-any` blocks until a signal by any process occurs.

Function `signal(`$P$`)` is implemented as follows. At lines **8**–**12**, the calling process $p$ uses its local *Work* array to create a "chain" of identifiers from $P$ (all elements of $Work[p][1..N]$ are initially $\perp$). This chain determines the order in which the processes in $P$ signal each other. To bootstrap the signalling mechanism, $p$ assigns `true` to $D[q][p]$ where $q$ is the process at the beginning of the chain (line **13**).

Function `wait(`$q$`)` is the counterpart of `signal(`$P$`)`. The argument of `wait(`$q$`)` is the ID of a process $q$ that may signal the caller. The process $p$ executing `wait` waits for a signal from $q$ by locally spinning on $D[p][q]$ (which is initially `false`) until a process that precedes it in a signalling chain writes `true` to $D[p][q]$ (see line **14**). At line **15**, $p$ reads the identifier of the next process in the chain (if any). If such a process exists, then $p$ signals it (lines **16**–**18**).

We also define a function `wait-any`, which takes no argument, and terminates when the calling process has been signalled through *any* signalling chain. In contrast to `wait`, this function does not signal the next process in the chain; a process must call `wait` subsequently for this to happen. The correctness properties of `signal`, `wait` and `wait-any`, are formally stated by the following lemma.

**Declarations** for `signal`, `wait` and `wait-any`.

**Shared variables**:

$Work[1..N][1..N]$ – array of process ID or $\perp$, initially all $\perp$, elements $Work[p][1..N]$ local to process $p$

$D[1..N][1..N]$ – array of Boolean, initially all $\mathsf{false}$, elements $D[p][1..N]$ local to $p$

**Private variables**: (per-process)

$prev$, $next, t$ – process ID or $\perp$, uninitialized

**Function** `signal(`$P$`)`

**Input**: $P \subseteq \mathcal{P}$

7 **if** $P = \emptyset$ **then return**

`// Create a ''signalling chain'' from elements of` $P$.

8 $prev := \perp$

9 **foreach** $next \in P$ **do**

`// Order of chain elements is reverse of loop order.`

10 **write** $Work[\mathsf{PID}][next] := prev$

11 $prev := next$

12 **end**

`// Signal the first process in the chain.`

13 **write** $D[next][\mathsf{PID}] := \mathsf{true}$

**Function** `wait(`$q$`)`

**Input**: $q \in \mathcal{P}$

14 **await** $D[\mathsf{PID}][q] = \mathsf{true}$

`// Identify next process in the signalling chain.`

15 $next :=$ **read**$(Work[q][\mathsf{PID}])$

16 **if** $next \neq \perp$ **then**

`// Signal the next process.`

17 **write** $D[next][q] := \mathsf{true}$

18 **end**

**Function** `wait-any()`

19 **loop forever**

`// Wait for signal from any process.`

20 **foreach** $t \in \mathcal{P}$ **do**

21 **if** $D[\mathsf{PID}][t] = \mathsf{true}$ **then**

22 **return**

23 **end**

24 **end**

25 **end**

Figure 4: Work-sharing signalling mechanism.

**Lemma 3.10** (safety)**.** *For any history:*

*(a) In the DSM model, each call to* `signal` *and* `wait` *incurs* $O(1)$ *RMRs, and each call to* `wait-any` *incurs zero RMRs.*
*(b) Each call to* `signal` *performs a bounded number of steps.*
*(c) If process* $p$ *completes a call to* `wait(`$q$`)` *then* $q$ *previously made a call to* `signal(`$P$`)` *with* $p \in P$*.*
*(d) If process* $p$ *completes a call to* `wait-any()` *then some process* $q$ *previously made a call to* `signal(`$P$`)` *with* $p \in P$*.*

*Proof.*
**Parts (a) and (b):** These follow directly from the algorithms and the locality of $Work[p][1..N]$ to process $p$.
**Part (c):** Suppose that $p$ completes a call to `wait(`$q$`)`. Then $p$ reads $D[p][q] =$ `true` at line **14**. Since this variable is initially `false`, some process $r$ must have assigned $D[p][q] =$ `true`, either at line **13** of `signal`, or at line **17** of `wait`. In the former case, it follows from the algorithm for `signal(`$P$`)` that $r = q$ and that $p \in P$. In the latter case, $r$ read $q$'s ID from $Work[q][r]$ at line **15**, which must have been written at line **10** of `signal(`$P$`)`, namely by $q$ with $p \in P$.
**Part (d):** The proof is a simplified version of the proof of part (c). □

**Lemma 3.11** (liveness)**.** *For any fair history:*

*(a) If some process* $q$ *writes* `true` *to* $D[p][q]$ *at line* **13** *or at line* **17**, *and if* $p$ *calls* `wait(`$q$`)`, *then* $p$*'s call terminates.*
*(b) If some process* $q$ *writes* `true` *to* $D[p][q]$ *at line* **13** *or at line* **17**, *and if* $p$ *calls* `wait-any`, *then* $p$*'s call terminates.*

*Proof.* **Part (a):** Follows directly from the fact that once a process assigns $D[p][q] =$ `true`, this variable is never reset back to `false`. Thus, any call to `wait(`$q$`)` by $p$ in a fair history eventually progresses beyond line **14** and terminates.
**Part (b):** The proof is analogous to the proof of part (a). □

**Lemma 3.12** (liveness)**.** *Consider any fair history* $H$ *where processes call the subroutines* `signal`, `wait` *and* `wait-any`. *Consider a particular call to* `signal(`$P$`)` *in* $H$, *say by some process* $q$. *Suppose that no other call by* $q$ *to* `signal(`$P'$`)` *occurs with* $P \cap P' \neq \emptyset$. *Also suppose that in* $H$ *every process* $p \in P$ *either makes a call to* `wait(`$q$`)` *or makes a non-terminating call to* `wait-any`. *Then all the calls to* `wait(`$q$`)` *and* `wait-any` *made in* $H$ *by processes in* $P$ *terminate.*

*Proof.* If $|P| = 0$ then the result follows trivially, so consider the case when $|P| \geq 1$. Let $\sigma = \langle p_1, p_2, ..., p_m \rangle$ be the sequence of process IDs selected at line **9** during $q$'s execution of `signal(`$P$`)` under consideration, in reverse order (i.e., $q$ assigns $D[p_1][q] =$ `true` at line **13**). Note that $\sigma$ is a sequence over all the elements of $P$ without repetition. Let $S(k)$ represent the following claim: all calls to `wait-any` and `wait` under consideration made by the first $k$ processes in $\sigma$ terminate. We will show that $S(k)$ holds for all $k \in [1..m]$ by induction on $k$, which implies the lemma.
**Basis:** $S(1)$ follows from Lemma 3.11 (a) and (b) since $q$ assigns $D[p_1][q] =$ `true` at line **13** during its call to `signal(`$P$`)`.
**Induction step:** Suppose that $m \geq 2$ (i.e., $|P| \geq 2$). For any $k \in [2..m]$, and for all $i \in [0..k-1]$, suppose that $S(i)$ holds. We must show that $S(k)$ holds. By the induction hypothesis, it suffices to show that the calls to `wait-any` and `wait(`$q$`)` by $p_k$ terminate. To that end, we will show that $p_k$ assigns $D[p_k][q] =$ `true` at line **17** of `wait(`$q$`)`. To see this, first note that by the induction

hypothesis and our assumption on when processes call `wait(q)`, $p_{k-1}$ eventually calls `wait(q)` and reads $Work[q][\mathsf{PID}]$ at line **15**. Since we assume $q$ does not call `signal(P')` with $P \cap P' \neq \emptyset$, it follows that $p_{k-1}$ reads $p_k$'s ID from $Work[q][\mathsf{PID}]$, and then assigns $D[p_k][q] = \mathsf{true}$ at line **17**. Consequently, any call to `wait-any` by $p_k$ terminates by Lemma 3.11 (b) and any call to `wait(q)` by $p_k$ terminates by Lemma 3.11 (a), as wanted. □

### 3.5 Name Consensus in the DSM Model: A Detailed Description

The name consensus algorithm (`NameDecide()`) that uses the instrumented leader election algorithm $\hat{L}$, handshaking protocol, and signalling mechanism described earlier, is presented in Figure 5. We will refer to the corresponding concurrent system (see Section 2) as $\mathcal{A}_{NC\text{-}DSM}$. The algorithm uses $N^2$ "instances" of a two-process leader election algorithm for handshaking, each instance having its own distinct copy of the underlying shared variables. We denote the instance for $p$ and $q$, which is local to $p$, by $L2P[p][q]$.

Leader election using $\hat{L}$ occurs at line **26**. For each process $p$, the computation of the sets $R_p$ and $W_p$ is performed implicitly by $\hat{L}$. We refer to $p$ and $q$ as *neighbours* if and only if $p$ and $q$ are adjacent (ignoring direction of edges) in the (directed) "data flow" graph $G$ defined in Section 3.1 (i.e., $p \in R_q \cup W_q$ or $q \in R_p \cup W_p$). As we prove later, the graph $G$ has useful connectedness properties.

After computing $R_p$ and $W_p$, process $p$ needs to communicate with its neighbours regarding the leader's ID, which is the response (i.e., winner) of the name consensus algorithm. As explained in Section 3.1, the high-level idea is to propagate this response through $G$, along the directed edges and away from the leader. We refer to this informally as propagating information "downstream" in $G$, even though $G$ may contain cycles. The leader's ID is itself stored in a global shared register *leader*, initially $\perp$. If $p$ is not the leader, then it attempts to discover the leader's ID using two mechanisms. First, it attempts to "pull" information from an upstream neighbour on a directed path from the leader to $p$. To that end, $p$ engages in the handshaking protocol described in Section 3.3 and tries to contact every neighbour in $R_p$. If $p$ fails to contact some such neighbour then that neighbour already knows the leader's ID and so $p$ can read this ID immediately from *leader*. On the other hand, if $p$ succeeds in contacting each neighbour then none of these neighbours knows the leader's ID and so $p$ waits for some neighbour (not necessarily one it has contacted) to "push" information to it. More precisely, "push" means that some neighbour signals $p$ and then $p$ reads *leader*. Finally, once $p$ discovers the leader's ID (through either the pull or push mechanism), it pushes information to all its neighbours in $W_p$, and to any neighbour that contacted $p$ using the handshaking protocol. For subtle reasons related to the work-sharing signalling mechanism, $p$ may have to perform additional work at this point to ensure that processes it contacted earlier make progress.

Now consider the outcome of executing line **26**. If $p$ is elected leader, then it writes its ID to *leader* at line **27**. At lines **29**–**33**, $p$ tries to contact its neighbours from $R_p$. (Since $R_p$ has $O(1)$ elements (see Lemma 3.7 (c)), this takes $O(1)$ RMRs in total.) Here $p$ stores in the set $U$ the IDs of neighbours that were actually contacted. If there is some $q \in R_p$ that $p$ fails to contact, then $q$ has progressed to line **41**, and so as we show in Lemma 3.15, *leader* $\neq \perp$. If $p$ does not win at line **26** and it contacts each $q \in R_p$, then $p$ still does not know the leader when it reaches line **34**. In this case, $p$ waits at line **35** for a signal from *any* neighbour by calling `wait-any()`.

By the time $p$ reaches line **37**, *leader* $\neq \perp$ holds, as we show later in Lemma 3.15. Now $p$ must signal some of its neighbours of this. The algorithm deals first with $p$'s downstream neighbours in $W_p$. To that end, $p$ calls `signal({q})` for each $q \in W_p$ at line **38**. Next, $p$ attempts to communicate with other neighbours downstream of it in $G$, namely any process $q$ for which $p \in R_q$ holds, which

**Declarations**

**Shared variables**: (global)

$\hat{L}$ – instrumented $O(1)$-RMR leader election algorithm (see Section 3.2)

*leader* – process ID or $\perp$, initially $\perp$

$L2P[1..N][1..N]$ – array of $O(1)$-RMR two-process LE algorithms, where $L2P[p][q]$ is for $p$ and $q$, and is local to $p$

**Private variables**: (per-process)

$R, W, U, X$ – sets of process IDs, initially $\emptyset$

$q$ – process ID, uninitialized

**Function** `NameDecide()`

**Output**: PID of leader

```
// Execute ''instrumented'' leader election algorithm.
26 if L̂.LeaderElect() = win then
27 |  write leader := PID
28 end
// Note: sets R and W have been computed (implicitly) at line 27.
// Try to contact neighbours in R.
29 foreach q ∈ R do
30 |  if L2P[q][PID].LeaderElect() = win then
31 |  |  U := U ∪ {q}
32 |  end
33 end
// If needed, wait for a signal (that leader ≠ ⊥).
34 if read(leader) = ⊥ then
35 |  wait-any()
36 end
// Invariant: leader ≠ ⊥. Next, signal out-neighbours in W.
37 foreach q ∈ W do
38 |  signal({q})
39 end
// Discover other out-neighbours.
40 foreach q ∈ P \ {PID} do
41 |  if L2P[PID][q].LeaderElect() = lose then
42 |  |  X := X ∪ {q}
43 |  end
44 end
// Signal discovered neighbours except those already signalled.
45 signal(X \ W)
// Share work in signalling chains of neighbours contacted at line 30.
46 foreach q ∈ U do in parallel
47 |  wait(q)              // Note: loop body executed concurrently for all q.
48 end parallel
// All parallel calls to wait at line 47 completed by now. Ready to return.
49 return read(leader)
```

Figure 5: Name consensus algorithm for the DSM model.

means that $q$ tried to contact $p$ at line **30**. If $q$ did contact $p$, then $q$ may rely on $p$ to signal it when $leader \neq \perp$. Consequently, $p$ determines the IDs of all such processes at lines **40**–**44**, collects these IDs in the set $X$, and then signals them all at line **45**. Processes participating in this signalling chain share work, which is necessary since there may be so many of them that $p$ cannot directly communicate with all of them in a constant number of RMRs. The argument in $p$'s call to `signal` at line **45** is actually the set difference $X \setminus W$ and not $X$ itself, which is done for two reasons: First, is efficiency – since $p$ already called `signal`($\{q\}$) for each process $q \in W$ at line **38**, it is not necessary to signal these processes again. The second reason is to meet the conditions of Lemma 3.12 – process $p$ may not call `signal` twice with arguments that are non-disjoint sets, as this may break the two signalling chains.

Finally, $p$ does its share of the work for each of the signalling chains it entered by contacting a neighbour at line **30**. Each of these neighbours will try to signal $p$ that $leader \neq \perp$ by calling (via `signal`), and will rely on $p$ calling `wait` to assist in the signalling mechanism. Thus, $p$ calls `wait`($q$) for each $q \in U$ at line **47**. More precisely, $p$ executes a *parallel for loop* at lines **46**–**48**, which makes multiple calls to `wait` in parallel by interleaving the corresponding operations, say in round-robin fashion. (We introduce parallelism here only to facilitate exposition. We could equally well use a modified version of `wait` that waits on multiple processes, but such a subroutine is somewhat more difficult to specify and analyze.) Note that $p$ cannot wait for each $q \in U$ sequentially because if $p$ were to block inside a particular call to `wait`($q$), $p$ could prevent progress in the signalling chain corresponding to some other process in $U$, leading to deadlock. If $p$ reaches line **49**, then all the parallel calls have terminated, and $leader \neq \perp$ holds (and has held since $p$ reached line **37**). Thus, `NameDecide()` returns the leader's ID to $p$.

To prove the correctness of `NameDecide()` (see Theorem 3.19 at the end of this section), we first establish some technical lemmas.

**Lemma 3.13.** *Let $H$ be any history of $\mathcal{A}_{NC\text{-}DSM}$ in which Condition 3.1 holds, and every active process has completed the call to $\hat{L}$.`LeaderElect()` at line* **26**. *For any pair of distinct processes $p, q \in \mathcal{P}$, if $p$ reads a value written by $q$ while calling $\hat{L}$.`LeaderElect()` in $H$, then once $p$ completes this call, $q \in R_p$ or $p \in W_q$.*

*Proof.* If, while executing line **26**, $p$ reads remotely a value written by $q$, then $p$ adds $q$ to $R_p$. Otherwise, $p$ reads a value written by $q$ in $p$'s local memory, in which case $q$ adds $p$ to $W_q$, because $p$ and $q$ are distinct. □

**Lemma 3.14.** *Let $H$ be any history of $\mathcal{A}_{NC\text{-}DSM}$ where Condition 3.1 holds, and every active process has completed the call to $\hat{L}$.`LeaderElect()` at line* **26**. *Suppose that the call to $\hat{L}$.`LeaderElect()` that ends last does so in step $i$ of $H$. Let $G_H$ denote the directed graph $(V, E)$ where $V$ is the set of process IDs and for any $p, q \in V$, $(p, q) \in E$ iff $p \in R_q$ or $q \in W_p$ in state $H[i]$ (and thereafter). Let $l$ denote the process elected leader at line* **26**. *Then, every process $p$ active in $H$ is reachable from $l$ in $G_H$.*

*Proof.* Suppose, by way of contradiction, that there is some process $p$ active in $H$ that is not reachable from $l$ in $G_H$. Let $R$ be the set of vertices of $G_H$ that are reachable from $l$, and let $\bar{R}$ be the remaining vertices. Note that $l \in R$ and $p \in \bar{R}$, so both sets are nonempty. Our key observation is that $H|\bar{R}$, like $H$, is a history where each process in $\bar{R}$ calls `NameDecide()` at most once. This is because by Lemma 3.13 and our definition of $\bar{R}$, no process $q \in \bar{R}$ reads (in $H$ or in $H|\bar{R}$) from a shared variable a value written by a process in $R$. (If this were false, there would be an edge in $G_H$ from a vertex in $R$ to a vertex in $\bar{R}$, contradicting our definition of these sets.) Moreover, $H|\bar{R}$ is a history of `NameDecide()` in which every active process completes its call to $\hat{L}$.`LeaderElect()` at

line **26**, and receives the same response as in $H$. It follows from Specification 3.5 and Lemma 3.7 (a) that this response in $H$ is lose for every process different from $l$, which contradicts Lemma 3.7 (a) for $H|\bar{R}$ because $l$ is not active in $H|\bar{R}$ by our definition of $\bar{R}$. (Since $l$ is reachable from itself, it belongs in $R$ and not $\bar{R}$.) □

**Lemma 3.15.** *Let $H$ be any history of $\mathcal{A}_{NC\text{-}DSM}$ where Condition 3.1 holds. If some process has reached line* **37** *in some prefix $H'$ of $H$, then $leader \neq \perp$ holds in $H$ after the prefix $H'$.*

*Proof.* First, note that *leader* is written at most once during $H$, namely at line **27** by the process elected leader using $\hat{L}$. Consequently, the property $leader \neq \perp$ is stable in $H$. It remains to show that for any process $p$, if $p$ reaches line **37** then $leader \neq \perp$. Suppose, for contradiction, that this is false; without loss of generality let $p$ be the *first* process to reach line **37** while $leader = \perp$ in $H$. A fortiori, $leader = \perp$ when $p$ reached line **34**, and therefore $p$ completed a call to `wait-any()` at line **35**. By Lemma 3.10 (d), some process $q$ previously called `signal(`$P$`)` with $p \in P$. Since $q$ calls `signal` only after it reaches line **37**, it follows that $q$ reached line **37** *before* $p$, and therefore while $leader = \perp$. This contradicts the definition of $p$ as the first process to reach line **37** while $leader = \perp$. □

**Lemma 3.16.** *Let $H$ be any history of $\mathcal{A}_{NC\text{-}DSM}$ where Condition 3.1 holds. The following hold in $H$:*

*(a) For any processes $p$ and $q$, the LE algorithm $L2P[p][q]$ is accessed according to Condition 3.4 and Condition 3.8.*

*(b) If $H$ is fair, all executions of* `signal` *at line* **45** *satisfy all the hypotheses of Lemma 3.12.*

*Proof.* **Part (a):** It suffices to show that the two-process LE algorithm $L2P[p][q]$ is executed at most once by $p$, at most once by $q$, and only with $p \neq q$. The only two places where $L2P[p][q]$ is executed are line **30** and line **41**. Thus, it follows from the loop conditions at line **29** and line **40** that if $L2P[p][q]$ is accessed, then $p \neq q$. (At line **30**, this is because $z \notin R_z$ for any process $z$, by construction of the set $R$.) It also follows from the loops containing line **30** and line **41** that only $p$ and $q$ execute $L2P[p][q]$, at most once at each line. If $p$ or $q$ executes $L2P[p][q]$ at both lines, then this implies $p = q$, which contradicts our earlier observation.
**Part (b):** Consider a call to `signal(`$P$`)` made by some process $q$ at line **45**. To satisfy the hypotheses of Lemma 3.12, we must show two things: (1) $q$ does not make another call to `signal(`$P'$`)`, where $P \cap P' \neq \emptyset$; and (2) every process $p \in P$ eventually makes a call to `wait(`$q$`)` at line **47**, provided that any prior call it makes to `wait-any` at line **35** terminates. For (1), note that any other such call to `signal(`$P'$`)` by $q$ must have been at line **38**, and in that case $P'$ is a singleton set containing an element of $W_q$. Then it follows from the set subtraction $X \setminus W$ at line **45** that $P \cap P' = \emptyset$, as wanted. For (2), consider any $p \in P$. Note that since $p \in P$ and by part (a) of this lemma, $p$ must have contacted $q$ at line **30** by Specification 3.5 and the algorithm. Thus, $p$ is active. Furthermore, since $H$ is fair, it follows from Lemma 3.16 (a), Specification 3.6, Lemma 3.10 (b), and the algorithm that $p$ either makes a non-terminating call to `wait-any` at line **35**, or reaches line **46**. In the latter case, $p$ begins the for loop at lines **46**–**48**, where it makes a call to `wait(`$z$`)` for each $z \in U_p$. Since these calls are made in parallel, and since $q \in U_p$ (because $p$ contacted $q$ at line **30**), $p$ eventually makes a call to `wait(`$q$`)`, as wanted. (Note that if a non-parallel for loop was used, $p$ might become blocked in some other call to `wait` before calling `wait(`$q$`)`.) □

**Lemma 3.17.** *For any history $H$ of $\mathcal{A}_{NC\text{-}DSM}$ where Condition 3.1 holds, each call to* `NameDecide()` *terminates.*

*Proof.* Suppose, for contradiction, that `NameDecide()` does not terminate for a subset $B$ of processes that are active in $H$. By Lemma 3.7 (a), Specification 3.6, and fairness of $H$, every process active in $H$ eventually completes its call to $\hat{L}$.`LeaderElect()` at line **26**. Let $G_H$ be the directed graph corresponding to $H$, as defined in the statement of Lemma 3.14. Let $l$ be the process that is elected leader at line **26** during $H$. By Lemma 3.14, there is a directed path in $G_H$ from $l$ to every process in $B$.

Now choose any process $p \in B$. Since $H$ is fair, it follows from the structure of `NameDecide()` that $p$ makes progress until it makes a non-terminating call to `LeaderElect`, `signal`, `wait-any`, or `wait`. Every active process completes line **26**, as argued earlier. All executions of the two-process leader election algorithm also terminate by Lemma 3.16 (a) and Specification 3.6. All executions of `signal` terminate by Lemma 3.10 (b). Thus, it remains to rule out the following two cases:
**Case A:** $p$ makes a non-terminating call to `wait-any` at line **35**. Since $p$ reached line **35**, it follows from the algorithm, particularly lines **26**–**27** and the test at line **34**, that $p \neq l$. Now without loss of generality, assume that $p$ is chosen so that the length of the path from $l$ to $p$ in $G_H$ is minimal, and let $z$ be $p$'s upstream neighbour on this minimal path. Since there is an edge from $z$ to $p$ in this path, $z$ is active by our definition of $G_H$ (i.e., either $z$ wrote a variable local to $p$, or $p$ read a remote variable written by $z$). Moreover, $z \in R_p$ or $p \in W_z$.
**Subcase A-i:** $z \in R_p$. First, we will show that $p$ contacted $z$ at line **30**. Suppose otherwise. Then by the time $p$ evaluates the condition at line **34**, $z$ has already reached line **41** by Lemma 3.16 (a) and Specification 3.5. Consequently, Lemma 3.15 implies that $leader \neq \perp$ holds when $p$ is at line **34**. But this contradicts $p$ branching to line **35** as stated in Case A. Thus, $p$ contacts $z$, which implies (by Lemma 3.16 (a) and Specification 3.5) that $z$ loses $L2P[z][p]$ at line **41**. Consequently, $z$ adds $p$ to $X_z$ at line **42**, and then calls `signal(`$X_z \setminus W_z$`)` at line **45**, where $p \in X_z \setminus W_z$ unless $z$ already called `signal(`$\{p\}$`)` at line **38**. In either case, $p$'s call to `wait-any` at line **35** terminates by Lemma 3.12 and Lemma 3.16 (b), which contradicts the hypothesis of Case A.
**Subcase A-ii:** $p \in W_z$. Since $z$ is active, and since it terminates in $H$ by our selection of $p$, it follows that $z$ calls `signal(`$\{p\}$`)` at line **38**. In that case, $p$'s execution of `wait-any` at line **35** terminates by Lemma 3.11 (b), which contradicts the hypothesis of Case A.
**Case B:** $p$ makes a non-terminating call to `wait(`$q$`)` at line **47** for some process $q$. Note that $p$ contacted $q$ at line **30** since $q \in U_p$ when $p$ is at line **46**. Now consider $q$. Since $q \in R_p$, $p$ read a value written by $q$ in $H$, and so $q$ is active in $H$. Furthermore, $q \neq p$, and by our prior analysis (up to and including Case A), $q$ eventually reaches line **46**. Since $p$ contacted $q$ and $p \neq q$, it follows from Lemma 3.16 (a) and Specification 3.5 that $q$ loses $L2P[q][p]$.`LeaderElect()` at line **41** and so $p \in X_q$ when $q$ reaches line **46**. Consequently, either $p \in W_q$ and $q$ completed a call to `signal(`$\{p\}$`)` at line **38**, or $p \notin W_q$ and $q$ completed a call to `signal(`$X_q \setminus W_q$`)` at line **45** with $p \in X_q \setminus W_q$. In either case, $p$'s call to `wait(`$q$`)` at line **47** terminates by Lemma 3.12 and Lemma 3.16 (b), which contradicts the hypothesis of Case B. □

**Lemma 3.18.** *For any history $H$ of $\mathcal{A}_{NC\text{-}DSM}$ where Condition 3.1 holds, each call to* `NameDecide()` *incurs $O(1)$ RMRs in the DSM model.*

*Proof.* Let $p$ be any process that calls `NameDecide()`. We will show that $p$ performs $O(1)$ RMRs in this call. We consider each line of `NameDecide()` where $p$ may incur one or more RMRs, and argue that the number of RMRs $p$ incurs at each such line is $O(1)$.

- **lines 26–28**: The call to $\hat{L}$.`LeaderElect()` incurs $O(1)$ RMRs by Lemma 3.7 (b). At most one additional RMR occurs at line **27**.
- **lines 29–33**: The loop here has $|R_p|$ iterations, where $|R_p| \in O(1)$ by Lemma 3.7 (c). RMRs

may occur only at line **30**, and each execution of this line incurs $O(1)$ RMRs by the RMR complexity of the two-process leader election algorithm, and by Lemma 3.16 (a).

- **lines 34–36**: There is at most one RMR at line **34**, and the call to `wait-any` at line **36** incurs zero RMRs by Lemma 3.10 (a).
- **lines 37–39**: Each of these lines is executed $|W_p|$ times, where $|W_p| \in O(1)$ by Lemma 3.7 (c). Here RMRs occur only at line **38**, and each execution of `signal` incurs $O(1)$ RMRs by Lemma 3.10 (a).
- **lines 40–44**: The loop here has $N$ iterations. RMRs may occur only at line **41**, and in fact an execution of $L2P[p][q]$.`LeaderElect()` by any process $p$, for any $q$, incurs zero RMRs by Specification 3.9 and Lemma 3.16 (a).
- **line 45**: The call to `signal` here incurs $O(1)$ RMRs by Lemma 3.10 (a).
- **lines 46–48**: Each of these lines is executed $|U_p|$ times, which is $O(1)$ since $U_p \subseteq R_p$ by the algorithm, and since $|R_p| \in O(1)$ by Lemma 3.7 (c). Here RMRs occur only at line **47**, where each execution of `wait` incurs $O(1)$ RMRs by Lemma 3.10 (a).
- **line 49**: At most one RMR occurs here.

□

**Theorem 3.19.** *For any history $H$ of $\mathcal{A}_{NC\text{-}DSM}$ (Figure 5) where Condition 3.1 holds, $H$ satisfies Specifications 3.2 and 3.3. Furthermore, each call to* `NameDecide()` *in $H$ incurs $O(1)$ RMRs in the DSM model.*

*Proof.* Consider any history $H$ of $\mathcal{A}_{NC\text{-}DSM}$ where Condition 3.1 holds.
**Specification 3.2:** Suppose that each call to `NameDecide()` terminates in $H$. It follows from the algorithm and Lemma 3.15 that `NameDecide()` returns to each caller the ID of the process $l$ that wins $\hat{L}$ at line **26**. By Lemma 3.7 (a), $l$ is the ID of a process that called $\hat{L}$.`LeaderElect()` in $H$, hence also called `NameDecide()`, as wanted.
**Specification 3.3:** Suppose that $H$ is fair. Then each call to `NameDecide()` in $H$ terminates by Lemma 3.17.
**RMR complexity:** This follows directly from Lemma 3.18. □

# 4 Pseudo-Locks

In this section, we define a new building block called a *pseudo-lock*, which underlies the implementations presented in later sections. We then show how to construct pseudo-locks using name consensus, reads and writes at a cost of $O(1)$ RMRs per process in the CC and DSM models.

Informally, a pseudo-lock is similar to a "one-shot" mutex, where at most one process acquires the critical section [10]. In addition, in a pseudo-lock any process that fails to acquire the CS must wait for the process that succeeded to leave the CS. Formally, a pseudo-lock is an algorithm that consists of two functions: an *entry protocol*, denoted `Pseudo-Enter()`, and an *exit protocol*, denoted `Pseudo-Exit()`. The two functions must be accessed according to the following etiquette:

**Condition 4.1.**

*(a) Each process calls* `Pseudo-Enter()` *and* `Pseudo-Exit()` *at most once.*

*(b) A process can call* `Pseudo-Exit()` *only after completing a call to* `Pseudo-Enter()`*.*

The correctness properties of a pseudo-lock are captured in Specifications 4.2–4.3.

**Specification 4.2** (safety)**.** *For any history where Condition 4.1 holds:*

*(a) If* `Pseudo-Enter()` *terminates, it returns a Boolean. If* `Pseudo-Exit()` *terminates, it returns* OK*.*

*(b)* `Pseudo-Enter()` *returns* true *to at most one process.*

*(c) If* `Pseudo-Enter()` *returns* false *to some process, then some other process has completed a call to* `Pseudo-Enter()` *with response* true *and subsequently made a call to* `Pseudo-Exit()`*.*

**Specification 4.3** (liveness)**.** *For any fair history where Condition 4.1 holds:*

*(a) If at least one call to* `Pseudo-Enter()` *is made, then at least one such call terminates.*

*(b) If some process calls* `Pseudo-Enter()` *with response* true *and then completes a call to* `Pseudo-Exit()`*, then all calls to* `Pseudo-Enter()` *terminate.*

*(c) Every call to* `Pseudo-Exit()` *terminates.*

**Definition 4.4.** *We say that a process* acquires the pseudo-lock *if it makes a call to* `Pseudo-Enter()` *with response* true*. We say that a process* fails to acquire the pseudo-lock *if it makes a call to* `Pseudo-Enter()` *with response* false*.*

In the remainder of this section, we present $O(1)$-RMR pseudo-lock implementations for the CC and DSM models.

## 4.1 Pseudo-locks in the CC Model

A pseudo-lock is straightforward to implement in the CC model given an $O(1)$-RMR name consensus algorithm, such as the one described in Section 3. One implementation is presented in Figure 6. (A very similar algorithm can be devised using leader election instead of name consensus, but we use name consensus nevertheless for consistency with the DSM algorithm.) Let $\mathcal{A}_{PL\text{-}CC}$ denote the corresponding concurrent system.

**Declarations**

**Shared variables**: (global)

*flag* – Boolean, initially false

**Subroutines**: (global)

`NameDecide()` – $O(1)$-RMR name consensus algorithm (see Section 3)

**Private variables**: (per-process)

*winner* – process ID, uninitialized

**Function** `Pseudo-Enter()`

```
   Output: Boolean
50 winner := NameDecide()
51 if winner = PID then
52 |  return true
53 else
54 |  await flag = true
55 |  return false
56 end
```

**Function** `Pseudo-Exit()`

```
   // Note:  winner is assigned at
      line 50 of Pseudo-Enter.
57 if winner = PID then
58 |  write flag := true
59 end
60 return OK
```

Figure 6: Pseudo-lock for the CC model.

**Theorem 4.5.** *For any history $H$ of $\mathcal{A}_{PL\text{-}CC}$ where Condition 4.1 holds, Specifications 4.2 and 4.3 hold. Furthermore, each call to* `Pseudo-Enter()` *or* `Pseudo-Exit()` *incurs $O(1)$ RMRs in the CC model.*

*Proof.* The proof is straightforward and is deferred to Appendix A.1. □

## 4.2 Pseudo-lock Implementation for the DSM Model

A pseudo-lock is straightforward to implement in the DSM model given an $O(1)$-RMR name consensus algorithm, such as the one described in Section 3, multiple instances of a two-process leader election algorithm that can be made local to one process (see Section 3.3), and the functions `signal`/`wait` from Section 3.4. One implementation is presented in Figure 7. Let $\mathcal{A}_{PL\text{-}DSM}$ denote the corresponding concurrent system.

**Theorem 4.6.** *For any history $H$ of $\mathcal{A}_{PL\text{-}DSM}$ where Condition 4.1 holds, Specifications 4.2 and 4.3 hold. Furthermore, each call to* `Pseudo-Enter()` *or* `Pseudo-Exit()` *incurs $O(1)$ RMRs in the DSM model.*

*Proof.* As in the proof of Theorem 4.5, note that the name consensus algorithm is accessed according to Condition 3.1, and so we can appeal to Specifications 3.2 and 3.3. In particular, it follows easily from the algorithm and Specification 3.2 that the two-process LE algorithm instances are accessed (at line **65** and **72**) according to Conditions 3.4 and 3.8, and so we can appeal to Specifications 3.5, 3.6, and 3.9.

**Specification 4.2:** Properties (a) and (b) follow easily from the algorithm and Specification 3.2, as in the proof of Theorem 4.5. Now consider property (c). Suppose that `Pseudo-Enter()` returns false to some process $p$. Then some process $w \neq p$ won `NameDecide()` at line **61**. If process $p$ lost $L2P[w][p]$.`LeaderElect()` at line **65**, then $w$ won it at line **72** of `Pseudo-Exit()`. Otherwise,

**Declarations**

**Shared variables**: (global)

$L2P[1..N][1..N]$ – array of $O(1)$-RMR two-process LE algorithms, where $L2P[p][q]$ is for $p$ and $q$, and local to $p$ (see Section 3.3)

**Subroutines**:

`NameDecide()` – $O(1)$-RMR name consensus algorithm (see Section 3)

`signal`/`wait` – subroutines from Section 3.4

**Private variables**: (per-process)

*winner* – process ID, uninitialized

$q$ – process ID, uninitialized

$P$ – set of process ID, initially $\emptyset$

**Function** `Pseudo-Enter()`

```
   Output: Boolean
61 winner := NameDecide()
62 if winner = PID then
63 |  return true
64 else
65 |  if L2P[winner][PID].LeaderElect() =
   |  win then
66 |  |  wait(winner)
67 |  end
68 |  return false
69 end
```

**Function** `Pseudo-Exit()`

```
   // Note: winner is assigned at
   line 61 of Pseudo-Enter().
70 if winner = PID then
71 |  foreach q ∈ P \ {PID} do
72 |  |  if L2P[PID][q].LeaderElect() =
   |  |  lose then
73 |  |  |  P := P ∪ {q}
74 |  |  end
75 |  end
76 |  signal(P)
77 end
78 return OK
```

Figure 7: Pseudo-lock for the DSM model.

$p$ won at line **65** and then completed a call to `wait(`$w$`)` at line **66**, and so $w$ must have called `signal(`$P$`)` with $w \in P$ by Lemma 3.10 (c), namely at line **76** of `Pseudo-Exit()`. In either case, $w$ previously completed a call to `Pseudo-Enter()` with response `true` (by the success test at line **70**), and then called `Pseudo-Exit()`, as wanted.

**Specification 4.3:** Let $H$ by any fair history where Condition 4.1 holds. As in the proof of Theorem 4.5, property (a) holds since the winner of `NameDecide()` at line **61** completes its call to `Pseudo-Enter()`. Let $w$ denote this process. Next, consider property (b). Suppose that $w$ makes a call to `Pseudo-Exit()` after completing its call to `Pseudo-Enter()`. Any process $q \neq w$ that calls `Pseudo-Enter()` completes line **61** by Specification 3.3, and then begins executing lines **65**–**68** by the algorithm and Specification 3.2. The call to $L2P[w][q]$.`LeaderElect()` at line **65** terminates by Specification 3.6. For termination of the call to `wait(`$w$`)` at line **66**, it suffices to show that process $w$ calls `signal(`$P$`)` where $P$ is the set of process that call `wait(`$w$`)` at line **66** of `Pseudo-Enter()` (see Lemma 3.12). (The other hypothesis of Lemma 3.12 is that $w$ does not also call `signal(`$P'$`)` with $P \cap P' \neq \emptyset$, which holds since $w$ calls `signal` at most once.) To that end, note that by the algorithm, $P$ is the set of processes $z$ such that $z \neq w$ (see line **71**) and $w$ lost $L2P[w][z]$ at line **72**. Consequently, $z \in P$ if and only if $z$ wins $L2P[w][z]$ at line **65** by Specification 3.5, which occurs if and only if $z$ calls `wait` at line **66** (since $H$ is fair). The argument of the latter call is $w$ by the algorithm and Specification 3.2. Thus, $q$'s call to `wait` at line **66** terminates by Lemma 3.12. Finally, property (c) follows from the structure of `Pseudo-Exit()`, Specification 3.6, and the termination of `signal` (see Lemma 3.10 (b)).

**RMR complexity:** This follows from the structure of `Pseudo-Enter()` and `Pseudo-Exit()`, the RMR complexity of `NameDecide()` (given that Condition 3.1 holds), the RMR complexity of the two-process LE algorithms (given that Condition 3.4 holds), the locality of these algorithms (Specification 3.9), and the RMR complexity of `signal`/`wait` (Lemma 3.10 (a)). □

# 5 Block Manager

Our implementations of CAS and LL/SC manipulate data structures we call *blocks*. These are inspired by "cells" in Herlihy's universal wait-free construction [16], and are similar to the "blocks" used in the wait-free implementation of CAS by Luchangco, Moir and Shavit [20]. Blocks record the state of the target object, where the current state is stored in a specially designated *current block*. A process effects a state change using the "pointer-swinging" technique: it allocates a new block representing the new state, and designates that block as current. We say that a block is *fresh* if all its fields (i.e., the objects contained in it) are in their initial states. At initialization, a fresh block called the *initial block* is current.

Fresh blocks are allocated using a *block allocator*, denoted in our pseudo-code by the subroutine `AllocBlock()`. In this paper, we assume that this function returns a unique fresh block different from the initial block. Fresh blocks are drawn from an unbounded set, which can be maintained by each process using a private linked list and accessed using only local computation. (For a discussion of memory recycling, see the PhD thesis of Wojciech Golab [12].) Formally, we assume that calls to `AllocBlock()` incur $O(1)$ RMRs and satisfy the following properties:

**Specification 5.1** (safety)**.** *For any history:*

*(a) if a call to* `AllocBlock()` *returns response $x$ then $x$ is a block address never before returned by* `AllocBlock()` *and different from the initial block (of the block manager); and*
*(b) a call to* `AllocBlock()` *does not access any block.*

**Specification 5.2** (liveness)**.** *For any fair history, each call to* `AllocBlock()` *terminates.*

The current block is tracked using a typed shared object we call the *block manager*. The block manager type, $\tau_{BM} = (\mathcal{S}, s_{init}, \mathcal{O}, \mathcal{R}, \delta)$, is formally defined as follows. Each state in $\mathcal{S}$ is a tuple $(C, S)$ where $C$ denotes the address of the current block, and $S$ is a set of pairs of the form $(x, p)$, where $x$ is a block address and $p$ is a process ID. The initial state is $(b_0, \emptyset)$ where $b_0$ is the address of the initial block. The following operation types are defined: `getCurBlock()` and `chngCurBlock`$(x, y)$. The set of responses consists of the set of block addresses and the set of process IDs. The transition mapping is defined by the (atomic execution of) the pseudo-code shown in Figure 8. (Recall that a box around pseudo-code indicates a transition mapping.) Operation `getCurBlock()` returns the address of the current block. Operation `chngCurBlock`$(x, y)$ makes $y$ the current block, unless some process already called `chngCurBlock`$(x, ...)$, in which case it has no side-effect. (Here and subsequently we use "..." as a "wildcard" symbol.) We say that a `chngCurBlock` operation is *successful* in the first case and *failed* otherwise. A successful `chngCurBlock`$(x, y)$ returns the caller's ID. A failed `chngCurBlock`$(x, y)$ necessarily follows a successful `chngCurBlock`$(x, ...)$, and returns the ID of the process that executed the latter operation.

Note that, as specified, a successful `chngCurBlock`$(x, y)$ does not necessarily change the address of the current block from $x$ to $y$; it merely ensures that $y$ is current. However, later on when we use the block manager to implement CAS and LL/SC, we will call `chngCurBlock`$(x, y)$ in such a way that if it succeeds then it does change the current block from $x$ to $y$; see Lemma 6.6.

Finally, we require that any implementation of the block manager satisfy the following:

**Specification 5.3** (safety)**.** *Any history of the implementation is linearizable with respect to type $\tau_{BM}$.*

**Specification 5.4** (liveness)**.** *In any fair history of the implementation, each call to* `getCurBlock` *or* `chngCurBlock` *terminates.*

```
Function getCurBlock()
    Output: address of current block
79  return C
```

```
Function chngCurBlock(x, y)
    Input: x, y – blocks
    Output: ID of process whose call to
            chngCurBlock(x, ...) succeeded
80  if (x, p) ∈ S for some p then
81  |   return p
82  else
83  |   C := y
84  |   S := S ∪ {(x, PID)}
85  |   return PID
86  end
```

Figure 8: Definition of block manager operation types. (The current state is denoted by $(C, S)$.)

### 5.1 Linearizable $O(1)$-RMR Implementation

We now present a simple implementation of the block manager that uses only reads and writes, and has $O(1)$ RMR complexity in the CC and DSM models. We refer to this implementation as $I_{BM} = (\tau_{BM}, \mathcal{P}, \mathcal{B}, \mathcal{H})$. The implementation records the address of the current block in a shared register variable $D$. It also relies on an $O(1)$-RMR pseudo-lock, as described in Section 4. We use multiple "instances" of the pseudo-lock (one per block), each with its own copies of the underlying shared variables. The pseudo-lock is needed for synchronization inside `chngCurBlock`, namely when multiple processes apply `chngCurBlock(`*x, ...*`)` operations for some block $x$. In that case, processes use the pseudo-lock in block $x$ to decide whose operation will succeed, and to discover the ID of the process whose operation succeeded. The access procedures for `getCurBlock` and `chngCurBlock` are presented in Figure 9.

We will now show that the implementation $I_{BM}$ is linearizable, satisfies the termination property, and has $O(1)$ RMR complexity (see Section 2).

**Lemma 5.5.** *For any history $H$ of $I_{BM}$ and for any block $x$ accessed by any process in $H$:*

*(a) The pseudo-lock in block $x$ is accessed according to Condition 4.1.*
*(b) A read of $x \triangleright$ winner at line* **95** *of* `chngCurBlock` *returns the ID of the unique process that acquired the pseudo-lock in block $x$ and then completed line* **90** *during the same* `chngCurBlock` *operation execution.*

*Proof.* The proof is straightforward and is deferred to Appendix A.2. □

To prove linearizability, we define for each history $H$ of $I_{BM}$ a candidate linearization $\bar{H}$ as follows. First, for each operation execution on the target object in $H$, we assign a numerical "timestamp".

**Definition 5.6.** *The timestamp $s$ for an arbitrary operation execution $Op$ in $H$, say by process $p$, and its completion (where applicable), are defined as follows:*

**Operation type `getCurBlock()`:**

*(a) If $Op$ is complete in $H$ and $p$ reads $D$ at line* **87** *during $Op$ in step $i$ of $H$, then $s = i$.*

**Declarations**

**Shared variables**: (global)

$D$ – register, stores a block address, initially points to the initial block defined for $\tau_{BM}$

**Shared variables**: (per-block)

*winner* – register, stores a process ID or $\perp$, initially $\perp$

**Subroutines**: (one instance per-block)

`Pseudo-Enter()`/`Pseudo-Exit()` – $O(1)$-RMR pseudo-lock from Section 4

**Function** `getCurBlock()`

```
87 return read(D)
```

**Function** `chngCurBlock(`$x$, $y$`)`

```
88 if read(x ▷ winner) = ⊥ then
89 |  if x ▷ Pseudo-Enter() = true then
90 |  |  write D := y
91 |  |  write x ▷ winner := PID
92 |  |  x ▷ Pseudo-Exit()
93 |  end
94 end
95 return read(x ▷ winner)
```

Figure 9: Block manager implementation.

*(b) Otherwise $s$ is undefined, and $Op$ does not appear in $\bar{H}$.*

**Operation type** `chngCurBlock(`$x$, $y$`)`**:**

*(c) If $p$ writes $D$ at line* **90** *of* `chngCurBlock` *during $Op$ in step $i$ of $H$, then $s = i$. (The completion of $Op$, if $Op$ is pending in $H$, returns $p$'s ID.)*

*(d) Else if $Op$ is complete and $p$ does not write $D$ at line* **90** *during $Op$, and reads $x \triangleright winner$ at line* **95** *for some block $x$ during $Op$ in step $i$ of $H$, then $s = i$.*

*(e) Otherwise $s$ is undefined, and $Op$ does not appear in $\bar{H}$.*

To construct our candidate linearization $\bar{H}$ of $H$, we arrange operation executions for which timestamps are defined, in increasing order of timestamp. (The uniqueness of these timestamps follows easily from Definition 5.6.) Operation executions that are pending in $H$, and whose timestamps are defined, are completed as explained above.

**Lemma 5.7.** *$\bar{H}$ satisfies properties (a) and (b) of linearizability (sequential completion and order preservation, see Section 2).*

*Proof.* Property (a) follows from our construction of $\bar{H}$ and Definition 5.6. For property (b), note that by Definition 5.6, if the timestamp of an operation execution $Op$ by $p$ in $H$ is $i$, then $p$ executes step $i$ during $Op$ in $H$. Thus, if $Op$ and $Op'$ are operation executions in $H$ whose counterparts appear in that order in $\bar{H}$, then $Op$ has a smaller timestamp, and so either $Op$ and $Op'$ are concurrent in $H$, or $Op$ precedes $Op'$ in $H$. □

It remains to prove property (c) (conformity to type $\tau_{BM}$). To that end, we first define some useful notation. Let $Op_i$, $s_i$, and $p_i$ denote the $i$'th operation execution in $\bar{H}$ (counting from 1), its timestamp, and the calling process. If $Op_i$ is a `chngCurBlock` operation execution, then we will refer to the arguments of $Op_i$ as $x_i$ and $y_i$. Finally, let $\nu_i = (C_i, S_i)$ denote the state of the block manager after applying the first $i$ operation executions in $\bar{H}$ on a correctly implemented block manager initialized to $b_0$ (the initial block).

**Lemma 5.8.** *Implementation $I_{BM}$ satisfies property (c) of linearizability (conformity to type $\tau_{BM}$).*

*Proof.* Let $H$ be any history of $I_{BM}$. Since conformity to a type is a safety property it suffices to consider finite $\bar{H}$. Let $k = |\bar{H}|$. Define $s_0 = 0$ and $s_{k+1} = \infty$. We will prove that for any $i \in \mathbb{N}$, $0 \leq i \leq k$:

(a) For any integer $t \in [s_i, s_{i+1})$, $D = C_i$ holds in state $H[t]$.

(b) If $i > 0$, then the response of $Op_i$ is the correct response for an operation execution of that type applied in state $\nu_{i-1}$.

Part (b) implies the lemma, but we require both parts for induction. Now let $S(i)$ denote parts (a)–(b) for a particular value of $i$. Note that in $H$, the value of $D$ is changed only by an execution of line **90**, which is an atomic step that defines the timestamp of an operation execution (on the target object) in $\bar{H}$. Therefore, the value of $D$ does not change between atomic steps $s_i$ and $s_{i+1}$ in $H$. This, in turn, implies that to prove part (a) of $S(i)$, it suffices to prove that $D = C_i$ in state $H[s_i]$—and that is all we do in the inductive step that follows.

For $S(0)$, (a) follows from our earlier definition of $s_0$, and the initialization of $D$ to $b_0$ (the initial block). Part (b) holds trivially for $S(0)$. Now for any $i$, $0 < i \leq k$, suppose that $S(i-1)$ holds, and consider $S(i)$. We proceed by cases on how the timestamp $s_i$ was obtained.

**Case A:** $Op_i$ is a `getCurBlock` operation execution and process $p_i$ reads $D$ at line **87** in step $s_i$ of $H$. In this case, $\nu_i = \nu_{i-1}$.

$S(i)$ part (a) follows from $S(i-1)$ part (a) because $\nu_i = \nu_{i-1}$ and step $s_i$ in $H$ does not write $D$.

For $S(i)$ part (b), note that by the algorithm, $Op_i$ returns in $\bar{H}$ the value $p_i$ that read from $D$ in step $s_i$. By $S(i-1)$ part (a), this value equals $C_{i-1}$, which is the correct response for $Op_i$.

**Case B:** $Op_i$ is a `chngCurBlock(`$x_i$`, `$y_i$`)` operation execution and process $p_i$ wrote $D$ at line **90** in step $s_i$ of $H$.

Since $p_i$ acquires the pseudo-lock in block $x_i$ during the counterpart of $Op_i$ in $H$, it follows from Lemma 5.5 (a) and Specification 4.2 (b) that no other process does so in $H$. Furthermore, by Lemma 5.5 (a) and Specification 4.2 (c), no process completes a call to $x_i$`▷Pseudo-Enter()` at line **89** before step $s_i$ in $H$. Consequently, it follows from Definition 5.6 and our construction of $\bar{H}$ that $Op_i$ is the first `chngCurBlock(`$x_i$`, ...)` operation execution in $\bar{H}$. Thus, $Op_i$ succeeds, and so $\nu_i = (C_i, S_i)$ where $C_i = y_i$ and $S_i = S_{i-1} \cup \{(x_i, p_i)\}$.

$S(i)$ part (a) follows by the action of step $s_i$ in $H$, where $p_i$ writes $y_i = C_i$ to $D$.

For $S(i)$ part (b), we must show that $Op_i$ returns $p_i$'s ID since it succeeds. But this follows from line **95** and Lemma 5.5 (b).

**Case C:** $Op_i$ is a complete `chngCurBlock(`$x_i$, $y_i$`)` operation execution and process $p_i$ reads $x_i \triangleright$ *winner* at line **95** in step $s_i$ of $H$, but does not write $D$ at line **90** during the counterpart of $Op_i$ in $H$.

Since $p_i$ does not acquire the pseudo-lock in block $x_i$ and completes line **89** during the counterpart of $Op_i$ in $H$, we will show that some `chngCurBlock(`$x_i, ...$`)`, precedes $Op_i$ in $\bar{H}$. If $p_i$ reads $x \triangleright$ *winner* $\neq \perp$ at line **88**, then some process wrote $x \triangleright$ *winner* at line **91** before step $s_i$ in $H$. This happens during the counterpart of some `chngCurBlock(`$x_i, ...$`)` operation execution $Op_j$ in $H$ that falls under Case B above, and precedes $Op_i$ by the order of lines **90**–**91**, by Definition 5.6, and by our construction of $\bar{H}$. On the other hand, if $p_i$ executes line **89** and fails to acquire the pseudo-lock, then it follows from Lemma 5.5 (a) and Specification 4.2 (c) that some other process $q$ does acquire the same pseudo-lock, and then makes a call to $x_i \triangleright$ `Pseudo-Exit()` before step $s_i$ in $H$. This happens during the counterpart of some `chngCurBlock(`$x_i, ...$`)` operation execution $Op_j$ that falls under Case B above, and precedes $Op_i$ by Definition 5.6 and by our construction of $\bar{H}$. Thus, $Op_i$ is a failed `chngCurBlock`, and so $\nu_i = \nu_{i-1}$.

$S(i)$ part (a) follows from the fact that $\nu_{i-1} = \nu_i$ and the action of step $s_i$ in $H$, which does not overwrite $D$.

For $S(i)$ part (b), we must show that $Op_i$ returns the ID of the process that applies a successful `chngCurBlock(`$x_i, ...$`)` in $\bar{H}$. Recall that $Op_i$ returns the ID $p_i$ reads from $x_i \triangleright$ *winner* at line **95**, which by Lemma 5.5 (b) is the ID of the unique process $q$ that acquired the pseudo-lock in block $x_i$ and then executed line **90**. The corresponding `chngCurBlock(`$x_i, ...$`)` operation execution by $q$ appears in $\bar{H}$ by Definition 5.6 and our construction of $\bar{H}$, and is successful by our analysis of Case B, as wanted.

□

**Theorem 5.9.** *The implementation $I_{BM}$ of the block manager satisfies Specifications 5.3 and 5.4. Furthermore, for any history $H$ of $I_{BM}$, each call to* `getCurBlock` *or* `chngCurBlock` *incurs $O(1)$ RMRs in the CC and DSM models.*

*Proof.* Let $H$ be any history of $I_{BM}$.
**Specification 5.3:** Linearizability of $H$ follows from Lemma 5.7 and Lemma 5.8 (b).
**Specification 5.4:** If $H$ is fair, each call to `getCurBlock` terminates by the structure of the access procedure. Similarly, each call to `chngCurBlock` terminates provided that the pseudo-lock functions `Pseudo-Enter` and `Pseudo-Exit` terminate. For termination of the latter functions, first recall that by Lemma 5.5 (a), Condition 4.1 holds with respect to any pseudo-lock accessed in $H$. Since Condition 4.1 holds, and since any process that acquires a pseudo-lock in any block $x$ eventually calls $x \triangleright$ `Pseudo-Exit()`, the pseudo-lock functions terminate by Specification 4.3.
**RMR complexity:** The RMR complexity of $I_{BM}$ follows from the structure of the access procedures and the RMR complexity of the pseudo-lock functions (where Condition 4.1 holds by Lemma 5.5 (a)). □

# 6 Extended Compare-and-Swap

Our goal in this section is to provide $O(1)$-RMR implementations, using reads and writes only, of two well-known synchronization primitives: compare-and-swap (CAS) and load-linked/store-conditional (LL/SC). We first give precise definitions of these primitives as shared object types, and then describe our implementation methodology. Our definitions at this stage are simplified in the sense that the type for each primitive does not support a `Write` operation type; we defer discussion of writable objects to Section 8.

We model the CAS primitive as a shared object type $\tau_{CAS}$. The set of states of the type is the set of values from a domain U, and the initial state $s_{init}$ can be any element of U. The following operation types are defined: `CAS(`*cmp*, *new*`)` and `Read()`. The set of responses is also U. The transition mapping is defined by the (atomic execution of) the pseudo-code shown in Figure 10. `CAS(`*cmp*, *new*`)` returns the prior state, which is a value from U, and also acts on the state according to one of two execution paths: A *successful* `CAS` operation occurs when the prior state is *cmp*, in which case it changes the state to *new*. A *failed* `CAS` operation occurs when the prior state is different from *cmp*, in which case the state does not change. `Read()` simply returns the previous state and does not change the state.

```
Function Read()
    Output: current value
96  return V
```

```
Function CAS(cmp, new)
    Input: cmp – comparison value
    Input: new – value to be swapped in
    Output: prior value
97  old := V
98  if old = cmp then
99  |   V := new
100 |   return old        // Operation successful.
101 else
102 |   return old          // Operation failed.
103 end
```

Figure 10: Definition of operation types for type $\tau_{CAS}$. (The current state is denoted by $V$.)

Load-linked/store-conditional (LL/SC) is another popular synchronization primitive, and is similar in spirit to CAS. We model LL/SC as a shared object type $\tau_{LL/SC}$ whose state consists of a value $V$ from a domain U and a Boolean array $Linked[1..N]$. In the initial state $s_{init}$, $V$ can be any element of U, and each element of $Linked[1..N]$ is false. The following operation types are defined: `LL()`, `SC(`*new*`)` and `Read()`. The set of responses consists of the elements of U (for `LL` and `Read`) and the Boolean constants {true, false} (for `SC`). The transition mapping is defined by (the atomic execution of) the pseudo-code shown in Figure 11. `LL` simply sets $Linked[\mathsf{PID}]$ and returns the current value. `Read` returns the current value and does not change the state, as in the case of type $\tau_{CAS}$. `SC(`*new*`)` has two execution paths: A *successful* `SC` operation occurs when in the prior state $Linked[\mathsf{PID}]$ = true; it changes $V$ to *new*, resets $Linked[1..N]$ to false, and returns true. A *failed* `SC` operation occurs when in the prior state $Linked[\mathsf{PID}]$ = false; it does not change the state and returns false.

Both CAS and LL/SC are important and commonly implemented (in hardware) primitives. (Popular architectures such as x86, Itanium, Sparc, MIPS, IBM Power, and DEC Alpha support a variant of either CAS or LL/SC.) They are typically used on a shared variable by calling `Read`

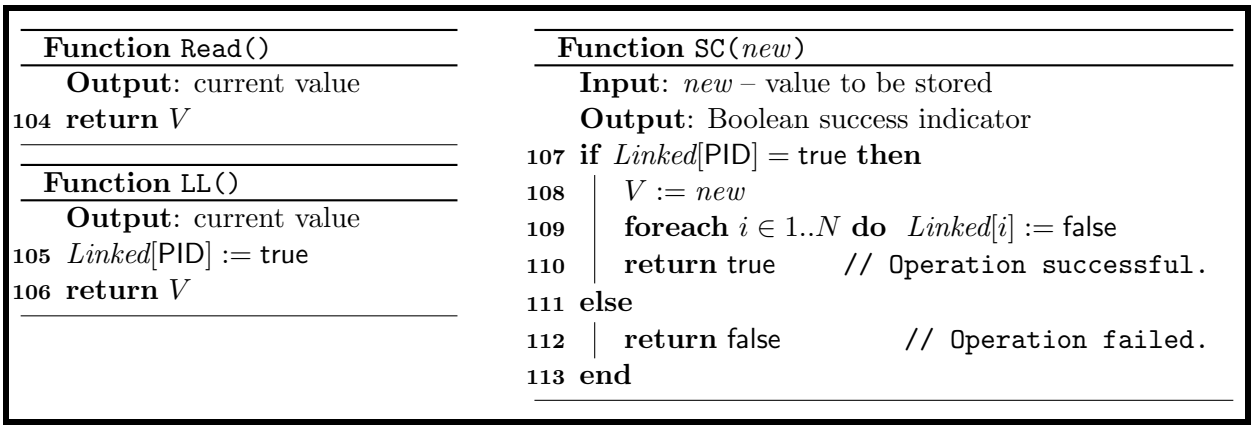


Figure 11: Definition of operation types for type $\tau_{LL/SC}$. (The current state is denoted by $V$ and $Linked[1..N]$.)

or LL first to retrieve the value, say $v$, and then calling CAS or SC to try changing the value from $v$ to some $v'$. In some applications, LL/SC is preferred over CAS because it does not suffer from the so-called *A-B-A problem*. That is, SC can "detect" when the value of an object has changed (say from A to B) since the last call to LL, and then changed back (from B to A), whereas CAS only "looks" at the latest value and succeeds or fails accordingly.

Although CAS and LL/SC are similar in spirit, due to the subtle difference between them, it is difficult to simulate one from the other in a manner that preserves the RMR-related correctness properties under consideration in this paper. (Constant-time wait-free simulations of LL/SC from CAS and vice-versa are known, e.g. [22, 14], but are insufficient for our purposes because they either use registers of unbounded size or they do not provide special "locality properties" defined later on in Section 7.) Rather than showing how to implement each type separately, we show how to implement a stronger object type called *extended compare-and-swap* (ECAS), from which CAS and LL/SC can be derived very easily. This new type, denoted $\tau_{ECAS}$, provides an operation type ECAS that behaves either like CAS or like SC depending on the value of a Boolean parameter (called *isSC*). Like CAS and SC, ECAS either succeeds or fails, which is indicated in its response.

The state space and initial value of $\tau_{ECAS}$ are defined as for $\tau_{LL/SC}$ (i.e., the combination of a value $V$ and a Boolean array $Linked[1..N]$). There are three operation types: Read, LL and ECAS(*isSC*, *cmp*, *new*). The set of responses consists of the elements of U (for LL and Read), as well as the set of ordered pairs of the form $(v, b)$ where $v \in$ U and $b \in \{\text{true}, \text{false}\}$ (for ECAS). The transition mapping is defined by (the atomic execution of) the pseudo-code shown in Figure 11 (for LL and Read) and in Figure 12 (for ECAS). ECAS is the new operation type that generalizes SC and CAS, and corresponds to the atomic execution of the pseudo-code shown in Figure 12. ECAS can simulate either SC or CAS depending on the value of the parameter *isSC*, as we explain below. It returns a pair $(v, b)$ consisting of the prior value $v$ and a Boolean success indicator $b$, which we also explain below.

The conditional statement in Figure 12 has two cases. In the first case, ECAS behaves either like a failed SC(*new*) operation (with $Linked[\text{PID}] = \text{false}$) or like a failed CAS(*cmp*, *new*) operation (with $cmp \neq V$), leaving $V$ and $Linked[1..N]$ unchanged. In the second case, ECAS behaves like a successful CAS(*cmp*, *new*) or SC(*new*) operation, assigning $V = new$ and also resetting $Linked[1..N]$. In both cases, the response is a tuple containing the prior value and a success indicator; we say that an ECAS operation is *successful* in the second case, and *failed* otherwise. (Note: At line **119** of Figure 12, it is not always necessary to reset *all* elements of $Linked[1..N]$, but we do so anyway

**Function** ECAS(*isSC, cmp, new*)

**Input**: *isSC* – Boolean, *cmp* – comparison value, *new* – value to be swapped in
**Output**: pair (prior value, Boolean success indicator)

```
114 old := V
115 if (isSC = true ∧ Linked[PID] = false) ∨ (isSC = false ∧ cmp ≠ old) then
116 |   return (old, false)                                   // Operation failed.
117 else
118 |   V := new
119 |   foreach i ∈ 1..N do Linked[i] := false
120 |   return (old, true)                                    // Operation successful.
121 end
```

Figure 12: Definition of ECAS operation type for type $\tau_{ECAS}$. (The current state is denoted by $V$ and $Linked[1..N]$.)

for simplicity. RMR complexity is not relevant in this context since we are defining the transition mapping for a shared object type rather than implementing that type.)

Implementing CAS and LL/SC (individually) is straightforward given a single ECAS base object. These implementations, which have the same RMR complexity (asymptotically) as the underlying ECAS object, are presented in Figure 13. In general, our implementations will satisfy the following properties:

**Specification 6.1** (safety)**.** *Any history of the implementation is linearizable with respect to the primitive's type.*

**Specification 6.2** (liveness)**.** *In any fair history of the implementation, each call to an access procedure terminates.*

The correctness properties of the implementations presented in Figure 13 are captured by the following theorem:

**Theorem 6.3.** *Let $\tau$ be one of $\tau_{CAS}$ or $\tau_{LL/SC}$. The implementation $I$ of $\tau$ presented in Figure 13 satisfies the following correctness properties:*

*(a) Specification 6.1 (linearizability with respect to type $\tau$).*
*(b) Specification 6.2 (termination).*
*(c) Operation executions on the target object have the same worst-case RMR complexity (asymptotically) as atomic steps on the base object $B$.*

*Proof.* Specification 6.1 (linearizability) follows easily if we consider that an operation execution on the target object takes effect at the same point as the corresponding atomic step on the base object $B$. Specification 6.2 (termination) and RMR complexity follow directly from the structure of the access procedures, each of which applies only a single atomic step, namely on the base object $B$. □

In the remainder of this section, we present a $O(1)$-RMR implementation of ECAS using reads and writes only. The implementation is linearizable but only in certain histories; this restriction simplifies the underlying algorithms, but at the same time allows the ECAS object to be used

**Declarations**

**Shared variables**: (global)

$B$ – ECAS object

**Private variables**: (per-process)

$val$ – value from domain U, uninitialized

$succ$ – Boolean, uninitialized

**Function** `Read()`

**Output**: current value

122 **return** $B$.`Read()`

**Function** `LL()`

**Output**: current value

125 **return** $B$.`LL()`

**Function** `CAS(`*cmp, new*`)`

**Input**: *cmp* – comparison value

**Input**: *new* – value to be swapped in

**Output**: prior value

123 $(val, succ) :=$ $B$.`ECAS(`false, *cmp, new*`)`

124 **return** *val*

**Function** `SC(`*new*`)`

**Input**: *new* – value to be stored

**Output**: Boolean success indicator

126 $(val, succ) :=$ $B$.`ECAS(`true, *new, new*`)`

127 **return** *succ*

Figure 13: Implementations of CAS and LL/SC from ECAS.

for implementing CAS and LL/SC (individually), as shown in Figure 13. The particular histories under consideration are those satisfying the following condition:

**Condition 6.4.** *Either no process invokes* `LL`*, or no process invokes* `ECAS(`*isSC, cmp, new*`)` *with* $isSC =$ false.

## 6.1 Linearizable $O(1)$-RMR Implementation of ECAS

We now present an implementation $I_E = (\tau_{ECAS}, \mathcal{P}, \mathcal{B}, \mathcal{H})$ of ECAS that uses a block manager, as described in Section 5, as its principal building block. The implementation uses blocks to record the state of the target object. To that end, each block contains fields called $V$ and $Linked[1..N]$, which correspond to the two components of the target object's state. Whenever a successful `ECAS` operation execution occurs, the caller allocates a new block and makes that block current by calling `chngCurBlock`. The latter operation execution performs much of the synchronization needed to handle concurrent `ECAS` operation executions by deciding which operation execution will succeed and which will fail. Two other fields are present in each block. First, an array $NextVal[1..N]$ is used (in some cases) by a successful `ECAS` operation execution to communicate the new value of the target object to a failed `ECAS` operation execution that must return that value. Second, a register *writer* is used to determine the ID of the process that allocated and made current a particular block.

The access procedures for the operation types `Read`, `LL`, and `ECAS` are presented in Figures 14–16. Lines containing shaded statements can be ignored safely for now; these statements come into play in Section 7 when we discuss locally-accessible implementations. For completeness, we provide in Figure 16 stub implementations of the subroutines called in shaded statements.

The access procedures for `Read`, `LL`, and `ECAS` are designed around the following invariant: the current state of the target object is stored in the variables $V$ and $Linked[1..N]$ in the current block. The access procedure for `Read` simply retrieves the current block, say $x$, and returns the value read

**Declarations** for ECAS implementation.

**Shared variables**: (global)

$M$ – $O(1)$-RMR block manager (see Section 5)

**Subroutines**: (global)

`AllocBlock()` – $O(1)$-RMR block allocator

**Shared variables**: (per-block)

$V$ – register, stores value from domain $\mathsf{U}$, initialized to the initial value of type $\tau_{ECAS}$

$Linked[1..N]$ – array of Boolean, initially all $\mathsf{false}$, element $i$ private to process $i$

$NextVal[1..N]$ – array of registers, same type as $V$, uninitialized

$writer$ – register, stores process ID or $\perp$, initially $\perp$

**Private variables**: (per-process)

$old$ – value from domain $\mathsf{U}$, uninitialized

$ret$ – Boolean, uninitialized

$d, d'$ – block addresses, uninitialized

$winner$ – process ID, uninitialized

**Function** `Read()`

```
128 d := M.getCurBlock()
129 HelperBegin(d)
130 old := read(d ▷ V)
131 HelperEnd(d)
132 return old
```

**Function** `LL()`

```
133 d := M.getCurBlock()
134 HelperBegin(d)
135 old := read(d ▷ V)
136 if read(d ▷ Linked[PID]) = false then
137 |   write d ▷ Linked[PID] := true
138 end
139 HelperEnd(d)
140 return old
```

Figure 14: Implementation $I_E$ of ECAS – part 1.

**Function** ECAS(*isSC, cmp, new*)

```
141 d := M.getCurBlock()
142 HelperBegin(d)
143 old := read(d ▷ V)
144 if (isSC = true ∧ read(d ▷ Linked[PID]) = false) ∨ (isSC = false ∧ cmp ≠ old) then
        // Operation execution failed.
145     ret := false
146 else if isSC = false ∧ cmp = new then
        // Operation execution successful, does not change the state.
147     ret := true
148 else if HelperCC(d, new) = true then
149     // Operation execution successful, changes the state
        // without changing the current block.
150     ret := true
151 else
        // Try to execute successful operation execution that changes the state.
152     d' := AllocBlock()
153     write d' ▷ writer := PID
154     write d' ▷ V := new
155     write d ▷ NextVal[PID] := new
156     winner := M.chngCurBlock(d, d')
157     if winner ≠ PID then
            // Operation execution failed.
158         old := read(d ▷ NextVal[winner])
159         ret := false
160     else
            // Operation execution successful, changes the state.
161         ret := true
162     end
163 end
164 HelperEnd(d)
165 return (old, ret)
```

Figure 15: Implementation $I_E$ of ECAS – part 2.

**Function** HelperBegin(*d*)

```
    // Do nothing.
```

**Function** HelperEnd(*d*)

```
    // Do nothing.
```

**Function** HelperCC(*d, new*)

**Output**: Boolean

```
166 return false
```

Figure 16: Implementation $I_E$ of ECAS – part 3.

from $x \triangleright V$. LL is similar, but also ensures that $x \triangleright \mathit{Linked}[\mathsf{PID}] = \mathsf{true}$ holds. The access procedure for ECAS begins at lines **141**–**143** in the same way as Read and LL. Then, at lines **144**–**147**, certain special cases are tested, which allow ECAS to return without trying to change the current block; these are the cases when the operation execution fails, or succeeds without changing the state of the target object. Lines **148**–**150** do nothing in this version of the implementation; they come into play only in Section 7.2.

If the execution of ECAS reaches line **152**, then the calling process has a chance to execute a successful ECAS, which will change the state of the target object. Since there may be several such operation executions that access the same block $x$, we allow one of them to succeed, and force the rest to fail. (We can always do so thanks to lines **144**–**147**, which for this reason are not merely an optimization.) The successful operation execution is the one whose chngCurBlock($x$, ...) succeeds at line **156**, after the caller allocates and initializes a new block at lines **152**–**154**. This operation execution returns the value of its *cmp* argument and a success indicator of true. (Note that a successful chngCurBlock($x$, $y$) ensures that $y.\mathit{Linked}[1..N] = \mathsf{false}$ because this array in block $y$ has not been accessed since initialization to false. Thus, such a successful chngCurBlock is the counterpart of line **119** in Figure 12.)

Prior to calling chngCurBlock($x$, ...) at line **156**, each process records its argument *new* in its entry of the array $x \triangleright \mathit{NextVal}[1..N]$ at line **155**. By doing so, each process ensures that if its subsequent chngCurBlock($x$, ...) succeeds then the new value of the target object is already recorded in block $x$, and so it can be accessed easily at line **158** by processes whose chngCurBlock($x$, ...) fails. The ECAS operation executions of processes in the latter category fail and return this new value (as well as a success indicator of false) without any additional waiting.

### 6.1.1 Analysis

We begin our analysis by proving linearizability. Then, we consider the RMR complexity and termination properties.

**Lemma 6.5.** *For any history $H$ of implementation $I_E$, and for any block $x$, if $x$ is current at some point in $H$, and subsequently an atomic step involving a successful $M$.*chngCurBlock(...,...) *occurs, then $x$ is not current at any point after that step in $H$.*

*Proof.* Suppose otherwise, and consider the second time $x$ becomes current. (If $x$ is the initial block, we count initialization as the first time.) This happens by the action of a successful $M$.chngCurBlock(...,$x$) at line **156**, and $x$ is the value returned by AllocBlock() at line **152**. Consequently, by Specification 5.1, $x$ is not the initial block, and so the first time $x$ became current was also by a successful $M$.chngCurBlock(...,$x$), preceded by another call to AllocBlock() that returned $x$. Thus, two calls to AllocBlock() return $x$ in $H$, which contradicts Specification 5.1. □

**Lemma 6.6.** *For any history $H$ of implementation $I_E$, for any blocks $x$ and $y$, and for any positive $k \in \mathbb{N}$, if a successful $M$.*chngCurBlock($x, y$) *occurs in step $k$ of $H$, then $x$ is current in state $H[k-1]$.*

*Proof.* Let $b_i$ denote the $i$'th block that becomes current (the initial block being $b_1$), and note that if $i > 1$ then $b_{i-1}$ is current just before $b_i$ becomes current. Suppose for contradiction that the lemma is false, and consider the smallest $i > 1$ such that block $b_i$ becomes current in step $k$ of $H$ by way of a successful $M$.chngCurBlock($b, b_i$) where $b \neq b_{i-1}$. Since $b$ is the value returned by $M$.getCurBlock() before this chngCurBlock, it follows that $b = b_j$ for some $j < i$. Now consider the subhistory $H'$ of $H$ after this getCurBlock and before step $k$. Since $b \neq b_{i-1}$, it follows by the

minimality of $i$ that a successful $M$.`chngCurBlock(`$b, ...$`)` occurs in $H'$. Since another successful $M$.`chngCurBlock(`$b, ...$`)` follows that one, namely in step $k$ of $H$, this contradicts the specification of type $\tau_{BM}$. □

**Lemma 6.7.** *For any history $H$ of implementation $I_E$, for any process $p$, and for any block $x$, if $p$ reads $x \triangleright V$ at line* **143** *at some point in $H$, then no process $q \neq p$ overwrites $x \triangleright V$ after that point in $H$.*

*Proof.* Recall that a block $x$ can become current at most once by Lemma 6.5. Next, note that $x \triangleright V$ is only written at line **154** of `ECAS`, which occurs before $x$ becomes current by way of a successful $M$.`chngCurBlock(...,`$x$`)` at line **156** (if this ever happens). Thus, $x \triangleright V$ is never written after $x$ becomes current, and in particular no process $q \neq p$ writes $x \triangleright V$ after $p$ makes a call to $M$.`getCurBlock()` with response $x$ at line **141** (hence after $p$ reads $x \triangleright V$ at line **143**) in $H$. □

To prove linearizability, we now define for each $H \in \mathcal{H}$ a candidate linearization $\bar{H}$ as follows. For each operation execution in $H$ (complete or pending), we assign a "timestamp", which is a tuple of the form $(x, t, q)$, where $x$ is a block address, $t$ is an integer, and $q$ is a process ID or zero.

**Definition 6.8.** *The timestamp $s$ for an arbitrary operation execution $Op$ in $H$, say by process $p$, and its completion (where applicable), are defined as follows:*

**Operation types `Read()` or `LL()`:**

*(a) If $p$ executes $M$.`getCurBlock()` at line* **128** *or line* **133** *during $Op$, say with response $x$, and then reads $x \triangleright V$ at line* **130** *or line* **135** *in step $i$ of $H$, then $s = (x, i, 0)$. (If $Op$ is pending in $H$, its completion returns the value read from $x \triangleright V$.)*

*(b) Otherwise $s$ is undefined.*

**Operation type `ECAS(`*isSC, cmp, new*`)`:**

*(c) If $p$ executes $M$.`getCurBlock()` at line* **141** *during $Op$ with response $x$, reads $x \triangleright V$ at line* **143** *in step $i$ of $H$, and then executes line* **145** *or line* **147**, *then $s = (x, i, 0)$.*
*(If $Op$ is pending in $H$, its completion returns the value read from $x \triangleright V$ and the Boolean assigned to ret at either line* **145** *or line* **147** *is executed.)*

*(d) Else if $p$ executes a successful $M$.`chngCurBlock(`$b, x$`)` at line* **156** *during $Op$ in step $i$ of $H$, then $s = (x, i, 0)$.*
*(If $Op$ is pending in $H$, its completion returns the value read from $b \triangleright V$ and* `true`*.)*

*(e) Else if $Op$ is complete in $H$, and $p$ executes a failed $M$.`chngCurBlock(`$b, y$`)` at line* **156** *during $Op$, then letting $i$ denote the atomic step (by any process) in $H$ involving a successful $M$.`chngCurBlock(`$b, x$`)`, then $s = (x, i, p)$.*
*(Since $b$ is a value returned by $M$.`getCurBlock()` and then a failed $M$.`chngCurBlock(`$b, ...$`)` occurs in $H$, step $i$ exists and is uniquely defined by the specification of type $\tau_{ECAS}$ and Lemma 6.6. Note also that process IDs are numbered starting at one and so clauses (d) and (e) never yield the same timestamp.)*

*(f) Otherwise $s$ is undefined.*

It follows from the above definitions that each operation execution for which the timestamp is defined has a unique timestamp. To construct our candidate linearization $\bar{H}$, we take all operation executions for which the timestamp is defined, and arrange these in increasing order of their timestamp according to Definition 6.9. Operation executions that are pending in $H$ are completed, as described earlier, if their timestamps are defined, and are discarded from $\bar{H}$ otherwise.

**Definition 6.9.** *For timestamps $(x_1, t_1, q_1)$ and $(x_2, t_2, q_2)$, we say that $(x_1, t_1, q_1) < (x_2, t_2, q_2)$ if and only if one of the following holds:*

- *$x_1 \neq x_2$ and block $x_1$ was current before block $x_2$ in $H$*
  *(Note that we can determine which of $x_1$ and $x_2$ was current first by Lemma 6.5.)*
- *$x_1 = x_2$ and $t_1 < t_2$*
- *$x_1 = x_2$, $t_1 = t_2$, and $q_1 < q_2$*

In this ordering, certain groups of concurrent `ECAS` operation executions have timestamps that match in the first two positions, say $(x, t, ...)$. The operation execution with timestamp $(x, t, 0)$ is successful; the others, with timestamps of the form $(x, t, p)$ for some process ID $p$, fail. Since their timestamps differ only in the third position, all operation executions in this group appear to take effect at almost the same point in $H$ – at step $t$. That is, the successful `ECAS` takes effect by applying a successful `chngCurBlock` in step $t$, and the others take effect immediately after this. (Note that "behind the scenes", these failed `ECAS` operation executions busy-wait, using pseudo-locks, until the successful one has taken effect; see Figure 9.)

Next, we define some useful notation. Let $Op_i$, $s_i$, and $p_i$ denote the $i$'th operation execution in $\bar{H}$ (counting from 1), its timestamp, and the executing process. Also let $(isSC_i, cmp_i, new_i)$ denote the arguments of $Op_i$ in the case when $Op_i$ is an `ECAS` operation execution. We now prove that $\bar{H}$ satisfies all the properties of a linearization.

**Lemma 6.10.** *$\bar{H}$ satisfies property (a) of linearizability (sequential completion).*

*Proof.* This follows directly from our construction on $\bar{H}$. In particular, any operation execution that is complete in $H$ has its timestamp defined. □

**Lemma 6.11.** *If an operation execution $Op$ in $H$ has timestamp $s = (..., t, ...)$ then $Op$ is pending in state $H[t]$.*

*Proof.* The lemma follows immediately from Definition 6.8 unless $Op$ falls under clause (e). In the latter case, during $Op$ there is a call to $M$.`getCurBlock()` with response $b$ followed by a failed $M$.`chngCurBlock(`$b, ...$`)`, and step $t$ is the one in which a successful $M$.`chngCurBlock(`$b, ...$`)` occurs. It follows from the specification of type $\tau_{ECAS}$ and from Lemma 6.6 that this successful `chngCurBlock` is unique and occurs in $H$. Moreover, by Lemma 6.5 it must occur between the `getCurBlock` and failed `chngCurBlock` during $Op$. Thus, $Op$ is pending just after step $t$, as wanted. □

**Lemma 6.12.** *If an operation execution $Op$ in $H$ has timestamp $s = (x, t, ...)$ then block $x$ is current at some point during $Op$.*

*Proof.* If $s$ does not fall under Definition 6.8 (d)–(e) then the call to $M$.`getCurBlock()` during $Op$ returns $x$, and so $x$ is current at that point. If $s$ falls under Definition 6.8 (d) then the process that executes $Op$ makes $x$ current in step $t$, which occurs during $Op$ by Definition 6.11. If $s$ falls under Definition 6.8 (e) then $x$ becomes current in step $t$ as explained in the proof of Lemma 6.11. □

**Lemma 6.13.** *$\bar{H}$ satisfies property (b) of linearizability (order preservation).*

*Proof.* Consider two distinct operation executions $Op_i$ and $Op_j$ in $\bar{H}$, with timestamps $s_i = (x_i, t_i, ...)$ and $s_j = (x_j, t_j, ...)$ respectively, such that $Op_i$ precedes $Op_j$ in $H$. We must show that $s_i < s_j$.
**Case A:** $x_i = x_j$. It follows from Lemma 6.11 that $Op_i$ and $Op_j$ are pending in states $H[t_i]$ and $H[t_j]$, respectively. Since $Op_i$ precedes $Op_j$ in $H$, this implies that $t_i < t_j$. Since $x_i = x_j$ and $t_i < t_j$, $s_i < s_j$ by Definition 6.9.
**Case B:** $x_i \neq x_j$. It follows from Lemma 6.12 that $x_i$ and $x_j$ are current at some point during $Op_i$ and $Op_j$, respectively. Since we assume $Op_j$ precedes $Op_i$ in $H$ and since $x_i \neq x_j$, it follows that $x_i$ becomes current before $x_j$ in $H$. (The order in which blocks become current is well-defined by Lemma 6.5.) Thus, $s_j < s_i$ by Definition 6.9. □

**Lemma 6.14.** *For any history $H$ of implementation $I_E$ where Condition 6.4 holds, any process $p$, any block $x$, and any operation execution $Op_i$ in $\bar{H}$ by $p$, if the timestamp $s_i$ of $Op_i$ is of the form $(..., ..., 0)$, and $p$ receives response $v$ when it reads $x \triangleright Linked[p]$ during the counterpart of $Op_i$ in $H$, then $v =$ true if and only if there is an* LL *operation execution in $\bar{H}$ before $Op_i$ and after any* ECAS *that precedes $Op_i$ in $\bar{H}$ and returns $(..., \mathsf{true})$.*

*Proof.* It follows from the access procedures of $I_E$, and the initialization of $x \triangleright Linked[p]$ to false, that $p$ can only read true from $x \triangleright Linked[p]$ during $Op_i$ if it previously completed an LL operation execution during which $p$ assigned $x \triangleright Linked[p] =$ true at line **137**. It remains to show that this LL occurs after any ECAS that precedes $Op_i$ in $\bar{H}$ and returns $(..., \mathsf{true})$. Suppose otherwise. Let $Op_l$ be $p$'s LL, and let $Op_e$ be the first ECAS that occurs between $Op_l$ and $Op_i$ in $\bar{H}$ and returns $(..., \mathsf{true})$. Since $Op_l$ is an LL, it follows from Condition 6.4 that $Op_e$ has argument $isSC_e =$ true. Consequently, it follows from our definition of $Op_e$ and from the ECAS access procedure that a successful $M$.chngCurBlock occurs during the counterpart of $Op_e$ in $H$ that makes some block $y$ current, where $y \neq x$ by Lemma 6.5. Now consider the timestamps of $Op_l$, $Op_e$, and $Op_i$. For $Op_l$, $s_l$ is of the form $(x, ..., 0)$ by Definition 6.8 (a). For $Op_e$, $s_e$ is of the form $(y, ..., 0)$ by Definition 6.8 (d). Finally, the timestamp $s_i$ of $Op_i$ is either of the form $(x, ..., 0)$ or of the form $(y', ..., 0)$ for some block $y' \neq x$ by Definition 6.8.
**Case A:** $s_i$ is of the form $(x, ..., 0)$. Since $s_l$ is of the form $(x, ..., 0)$ and $s_e$ is of the form $(y, ..., 0)$, where $y \neq x$, as noted earlier, Definition 6.9 contradicts $Op_e$ occurring between $Op_l$ and $Op_i$ in $\bar{H}$.
**Case B:** $s_i$ is of the form $(y', ..., 0)$ for some block $y' \neq x$. It follows that $s_i$ falls under Definition 6.8 (d), and so a successful $M$.chngCurBlock$(x', y')$ occurs during the counterpart of $Op_i$ in $H$ for some $x'$. In particular, $x' = x$ by the algorithm and the assumption that $p$ accesses $x \triangleright Linked[p]$ during the counterpart of $Op_i$ in $H$. Consequently, by Lemma 6.5, $y'$ is the next block that becomes current after $x$. But in that case by our construction of $\bar{H}$, $Op_i$ and $Op_e$ are the same operation execution, which contradicts our definition of $Op_e$. □

**Lemma 6.15.** *If $H$ satisfies Condition 6.4 then $\bar{H}$ satisfies property (c) of linearizability (conformity to type $\tau_{ECAS}$).*

*Proof.* Let $H$ be any history of $I_E$ where Condition 6.4 holds. Since conformity to a type is a safety property it suffices to consider finite $\bar{H}$. Let $k = |\bar{H}|$. Let $x_i$ and $t_i$ denote the first two components of $s_i$. Define $x_0$ as the initial block, and $x_{k+1}$ as the current block at the end of $H$. Define $s_0 = (x_0, 0, 0)$ and $s_{k+1} = (x_{k+1}, \infty, 0)$. Let $\nu_i$ for $0 \leq i \leq k$ denote the state of a correctly implemented ECAS object after applying the first $i$ operation executions in $\bar{H}$. Let $\nu_i.V$ and $\nu_i.Linked[1..N]$ denote the two components of this state. We will show that for any $i \in \mathbb{N}$, $0 \leq i \leq k$:

(a) For $t = t_i$ and any integer $t \in [t_i, t_{i+1})$, $x_i \rhd V = \nu_i.V$ holds in state $H[t]$.

(b) If $i > 0$, then the response of $Op_i$ is the correct response for an operation execution of that type applied in state $\nu_{i-1}$.

Part (b) implies the lemma, but we require both parts for induction. Now let $S(i)$ denote parts (a)–(b) for a particular value of $i$. Note that in $H$, the current block and value of field $V$ in that block are changed only by an execution of line **156**, which is an atomic step that defines the timestamp of an operation execution (on the target object) in $\bar{H}$. Therefore, the current block and value of $V$ in that block do not change between atomic steps $t_i$ and $t_{i+1}$ in $H$. This, in turn, implies that to prove part (a) of $S(i)$, it suffices to prove that $x_i \rhd V = \nu_i.V$ in state $H[t_i]$—and that is all we do in the inductive step that follows.

For $S(0)$, part (a) follows from our earlier definition of $x_0$ as the initial block and $t_0 = 0$, as well as the initialization of $x_0 \rhd V$ to the initial value of type $\tau_{ECAS}$. Part (b) holds trivially for $S(0)$. Now for any $i$, $0 < i \leq k$, suppose that $S(i-1)$ holds, and consider $S(i)$. We proceed by cases on how $s_i = (x_i, t_i, ...)$ was obtained, noting that $x_i = x_{i-1}$ unless $s_i$ falls under Definition 6.8 (d).

**Case A:** $Op_i$ falls under Definition 6.8 (a). In this case, $s_i = (x_i, t_i, 0)$ for some $t_i$, and either $Op_i$ is a `Read` and $p_i$ reads $x \rhd V$ in step $t_i$ of $H$ at line **130**, or $Op_i$ is an `LL` and $p_i$ reads $x \rhd V$ in step $t_i$ of $H$ at line **135**.

$S(i)$ (a) follows from $S(i-1)$ (a) because $\nu_i = \nu_{i-1}$, $x_i = x_{i-1}$, and since step $t_i$ by $p_i$ does not change the current block or write $x_{i-1} \rhd V$.

For $S(i)$ (b), note that by the algorithm, $Op_i$ returns the value $p$ reads from $x_i \rhd V$ in step $t_i$, which equals $\nu_{i-1}.V$ by $S(i-1)$ (a) and the fact that $x_{i-1} = x_i$. Since $Op_i$ is a `Read` or `LL`, this response is correct in $\bar{H}$.

**Case B:** $Op_i$ falls under Definition 6.8 (c), and $p_i$ executes line **145** during the counterpart of $Op_i$ in $H$. In this case, $s_i = (x_i, t_i, 0)$ for some $t_i$, and $Op_i$ is an `ECAS` operation execution where $p_i$ reads $x_i \rhd V$ at line **143** in step $t_i$ of $H$.

As in Case A, it follows that $p_i$ reads the value $\nu_{i-1}.V$ from $x_{i-1} \rhd V$ at line **143**. Similarly, it follows from Lemma 6.14 that $p_i$ reads the value $\nu_{i-1}.Linked[p_i]$ from $x_{i-1} \rhd Linked[p]$ at line **144**. Consequently, $Op_i$ returns $(\nu_{i-1}.V, \mathsf{false})$ in $\bar{H}$. Furthermore, by the success of the test at line **144**, either $(isSC = \mathsf{true} \wedge \nu_{i-1}.Linked[p_i] = \mathsf{false})$ holds, or $(isSC = \mathsf{false} \wedge cmp_i \neq \nu_{i-1}.V)$ holds. Thus, $Op_i$ fails.

$S(i)$ (a) follows from $S(i-1)$ (a) as in Case A since $Op_i$ is a failed `ECAS` operation execution, and so $\nu_i.V = \nu_{i-1}.V$.

$S(i)$ (b) holds since $Op_i$ returns $(\nu_{i-1}.V, \mathsf{false})$ in $\bar{H}$ and since $Op_i$ is a failed `ECAS` operation execution.

**Case C:** $Op_i$ falls under Definition 6.8 (c), and $p_i$ executes line **147** during the counterpart of $Op_i$ in $H$. In this case, $s_i = (x_i, t_i, 0)$ for some $t$, and $Op_i$ is an `ECAS` operation execution where $p_i$ reads $x_i \rhd V$ at line **143** in step $t$ of $H$.

As in Case B, it follows that $p_i$ reads the value $\nu_{i-1}.V$ from $x_{i-1} \rhd V$ at line **143**, and the value $\nu_{i-1}.Linked[p]$ from $x_{i-1} \rhd Linked[p_i]$ at line **144**. Consequently, $Op_i$ returns $(\nu_{i-1}.V, \mathsf{true})$ in $\bar{H}$. Furthermore, by the success of the test at line **146**, $isSC = \mathsf{false}$, $cmp_i = new_i$, and $cmp_i$ equals the value read from $x_{i-1} \rhd V$, which is $\nu_{i-1}.V$. Thus, $Op_i$ is successful, and leaves $\nu_i.V = \nu_{i-1}.V$.

$S(i)$ (a) follows from $S(i-1)$ (a) because because $Op_i$ is a successful `ECAS` operation execution where $\nu_i.V = \nu_{i-1}.V$, and since step $t_i$ by $p_i$ does not change the current block or write $x_{i-1} \rhd V$.

$S(i)$ (b) holds since $Op_i$ returns $(\nu_{i-1}.V, \mathsf{true})$ in $\bar{H}$ and since $Op_i$ is a successful `ECAS` operation execution.

**Case D:** $Op_i$ falls under Definition 6.8 (d), and $p_i$ executes a successful $M$.`chngCurBlock(`$d, d'$`)` for some $d$ and $d'$ at line **156** during the counterpart of $Op_i$ in $H$. In this case, $s_i = (x_i, t_i, 0)$ for some $t_i$, $x_i = d'$, $x_{i-1} = d$ (by Lemma 6.6) and $Op_i$ is an `ECAS` operation execution.

Since $x_{i-1} = d$, it follows that $p_i$'s call to $M$.`getCurBlock()` during the counterpart of $Op_i$ in $H$ returns $x_{i-1}$, and so as in Case B $p_i$ reads the value $\nu_{i-1}.Linked[p]$ from $x_{i-1} \rhd Linked[p]$. Similarly, it follows from Lemma 6.7 and the algorithm that the value $p_i$ reads from $x_{i-1} \rhd V$ is the value of this shared variable in state $H[t_i - 1]$, which is $\nu_{i-1}.V$ by $S(i-1)$ (a). Consequently, $Op_i$ returns $(\nu_{i-1}.V, \mathsf{true})$ in $\bar{H}$. Furthermore, by the failure of the tests at line **144** and line **146**, it follows that either $(isSC = \mathsf{false} \ \wedge \ cmp_i = \nu_{i-1}.V \ \wedge \ cmp_i \neq new_i)$ holds, or $(isSC = \mathsf{true} \ \wedge \ \nu_i.Linked[p_i] = \mathsf{true})$ holds. In either case, $Op_i$ is successful.

$S(i)$ (a) follows by the action of step $t_i$ by $p_i$, which is a $M$.`chngCurBlock()` that makes block $x_i$ current, and where $x_i = new_i$ by $p_i$'s prior execution of line **154**. (It follows from the algorithm and Specification 5.1 that no process can overwrite this value until possibly after step $t_i$.)

$S(i)$ (b) holds since $Op_i$ returns $(\nu_{i-1}.V, \mathsf{true})$ in $\bar{H}$ and since $Op_i$ is a successful `ECAS` operation execution.

**Case E:** $Op_i$ falls under Definition 6.8 (e), and $p_i$ executes a failed $M$.`chngCurBlock(`$d$, ...`)` for some $d$ at line **156** during the counterpart of $Op_i$ in $H$. In this case, $s_i = (x_i, t_i, p_i)$ for some $t_i$ and $Op_i$ is an `ECAS` operation execution. Furthermore, there is an `ECAS` operation execution $Op_j$ in $\bar{H}$ that falls under Definition 6.8 (d), and whose timestamp is $s_j = (x_j, t_j, 0)$ where $x_j = x_i$, $t_j = t_i$, and $j < i$. Moreover, $p_j$ applies a successful $M$.`chngCurBlock(`$d$, $x_j$`)` during the counterpart of $Op_j$, in step $t_j$ of $H$.

Now consider the response of $Op_i$. Since $p_i$ completes line **159** during $Op_i$, note that $p_i$'s failed $M$.`chngCurBlock(`$d$, ...`)` at line **156** returns $p_j$'s ID. Next, $p_i$ reads $new_j$ from $x_i \rhd NextVal[p_j]$ at line **158** (since this read occurs after $p_j$'s successful `chngCurBlock`). Since $Op_j$ is a successful `ECAS` operation execution (see Case D), $new_j = \nu_j.V$ holds, and so $Op_i$ returns $(\nu_j.V, \mathsf{false})$ in $\bar{H}$.

Next, we will show that $Op_i$ fails and $\nu_{i-1}.V = \nu_j.V$. The latter point follows from $S(i-1)$ (a) and Definitions 6.8 and 6.9, which imply that either $j = i-1$, or the only operation executions between $Op_j$ and $Op_i$ in $\bar{H}$ are ones with timestamps of the form $(x_j, t_j, ...)$ (and fall under Definition 6.8 (e), like $Op_i$). It remains to show that $Op_i$ fails. If $isSC_i = \mathsf{true}$, then this

holds because $Op_j$ is the last successful `ECAS` operation execution in $\bar{H}$ before $Op_i$ and there is no `LL` operation execution by $p_i$ between $Op_j$ and $Op_i$ in $\bar{H}$ (by Definitions 6.8 and 6.9), and so $\nu_{i-1}.Linked[1..N] = \mathsf{false}$. Otherwise, $isSC_i = isSC_j = \mathsf{false}$ (by Condition 6.4), and so $cmp \neq new$ holds for both $Op_i$ and $Op_j$ by the failure of the test at line **146**. It suffices to show that $cmp_i \neq \nu_{i-1}.V$. To that end, since $\nu_{i-1}.V = \nu_j.V = new_j$ and $cmp_j \neq new_j$, it suffices to show that $cmp_i = cmp_j$. To show this, it suffices in turn to show that $p_i$ and $p_j$ read the same value from $d \triangleright V$, since this value equals $cmp$ in both operation executions by the failure of the test at line **144**. Suppose otherwise. (Once again, we give a general argument that is slightly more complex than one needed to deal with implementation $I_E$ alone, but is used later on in Section 7.2.) Then some process $q$ writes $d \triangleright V$ between $p_i$ and $p_j$ reading it. It follows that $q$ is not $p_i$ or $p_j$, because neither $p_i$ nor $p_j$ writes $d \triangleright V$ during the counterparts of $Op_i$ and $Op_j$ (respectively) in $H$, and neither $p_i$ nor $p_j$ accesses block $d$ again after completing $Op_i$ and $Op_j$ by Lemma 6.5, $p_j$'s successful `chngCurBlock(`$d, ...$`)`, and $p_i$'s failed `chngCurBlock(`$d, ...$`)`. However, $q \notin \{p_j, p_i\}$ contradicts Lemma 6.7.

Thus $Op_i$ fails and leaves $\nu_i.V = \nu_{i-1}.V$. $S(i)$ (a) follows from $S(i-1)$ (a) since $\nu_{i-1}.V = \nu_j.V$ holds and $t_j = t_i$. $S(i)$ (b) holds because $Op_i$ returns $(\nu_{i-1}.V, \mathsf{false})$ in $\bar{H}$ and since $Op_i$ is a failed `ECAS` operation execution.

□

**Theorem 6.16.** *The implementation $I_E$ satisfies Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 6.4. Furthermore, each operation execution on the target object incurs $O(1)$ RMRs in the CC and DSM models.*

*Proof.* Specification 6.1 under Condition 6.4 follows directly from Lemma 6.10, Lemma 6.13, and Lemma 6.15. Specification 6.2 follows from the structure of the access procedures and from Specification 5.2. RMR complexity follows from the structure of the access procedures, the RMR complexity of the base objects, including the block manager object $M$, and the RMR complexity of subroutine `AllocBlock()`. □

# 7 Locally-Accessible Implementations

Consider an algorithm $\mathcal{A}$ that accesses an ECAS object $E$. Our goal in this section is to show that one can "plug" our simulation of $E$ into $\mathcal{A}$ with at most a constant-factor increase in $\mathcal{A}$'s RMR complexity, and while preserving other important correctness properties. (See Section 9 for a discussion of these properties.) To achieve this, our implementation of $E$ must meet the following *locality properties*, in addition to worst-case $O(1)$ RMR complexity: (1) in the DSM model, a designated process $p_{special}$ (to which we say $E$ is local) may execute *any* operation execution on $E$ without incurring any RMRs; and (2) in the CC model, certain "in-cache" operation executions on $E$ incur no RMRs.

Since the first property above is straightforward to define, we now focus on formalizing the second property. To begin with, we must define precisely when an operation on a shared object provided in hardware is an RMR in the CC model. Our definition is very much tied in with the one given in Section 2 where the only primitives considered for accessing shared memory were reads and writes. In a multiprocessor that supports other types of shared objects in hardware, we must determine which operations cause RMRs and how many. (Recall from Section 2 that an operation is the application of an operation type to a shared object, and comes in two flavours: atomic operations represented by atomic steps and non-atomic operations represented by operation executions.) To that end, we classify operations of every shared object type as being either *read-like* or *write-like*, and apply our definition from Section 2 by treating read-like operations like reads and write-like operations like writes. (We classify operations and not operation types because two applications of the same operation type may behave differently.) But what should we classify as read-like and write-like? To a first approximation, an operation is write-like if it changes the state of the shared object, and read-like otherwise. One has to be careful about the interpretation of this statement, however, because in certain cases there is more than one natural way to classify a particular operation.

Consider an object of type $\tau_{CAS}$ (as defined in Section 6). For this type, it is natural to treat `Read` and failed `CAS` as read-like, and successful `CAS` as write-like. However, in the special case when the arguments of a `CAS` operation satisfy $cmp = new$, it is also natural to classify a successful `CAS` as read-like because the state transition it causes is trivial. Given such a choice, the safe thing to do is to classify the operation as read-like, because the more operations are classified as read-like, the stronger become the correctness properties of locally-accessible implementations, and the harder it becomes to construct them. In particular, the more operations are read-like, the fewer RMRs the access procedures of the implementation are permitted to incur. (In this particular case, we can also assume without loss of generality that `CAS` is never called with $cmp = new$, since such calls can be simulated using `Read`.)

Next, consider type $\tau_{LL/SC}$ (as defined in Section 6). Recall that, according to the definition in Section 6, this state consists of a value denoted $V$ and a Boolean array $Linked[1..N]$. Applying our convention to this definition we classify failed `SC` as read-like, successful `SC` as write-like, and `LL` as write-like. However, note that when process $p$ applies `LL`, despite writing $Linked[p]$ this operation has no effect on the component of state that is read by other processes because only $p$ reads $Linked[p]$. Consequently, the change of state need not be propagated by the coherence protocol, and an RMR need not occur unless the value $V$ is also loaded into the local cache. (A simple hardware implementation of `LL`/`SC` encodes $Linked[p]$ in the state of $p$'s cached copy of the shared object. An `LL` by $p$ sets this bit only in $p$'s cache, and a successful `SC` triggers an invalidation that clears this bit in all caches holding copies of the shared object.) Thus, it is also natural to treat `LL` as read-like, and once again it is safe for us to do so as it increases the burden on the implementation. (Another reason for treating `LL` as read-like is that another definition of $\tau_{LL/SC}$

exists where LL does not modify the shared state at all.)

Finally, consider type $\tau_{ECAS}$, which we defined in Section 6 to serve as a precursor for CAS and LL/SC. We classify operations on such an object as follows: Read, LL, and failed ECAS are read-like, and successful ECAS is write-like except when $isSC = \mathsf{false}$ and $cmp = new$. (In the latter case, there is more than one natural classification, but we are free to classify such operations as read-like, as explained earlier.)

Now consider a (linearizable) implementation $I$ of a shared object type $\tau$ using only reads and writes. We would like the implementation $I$ to have the following *RMR-preservation property* in the write-through or write-back CC model:

**Definition 7.1** (RMR-preservation property in the CC model)**.** *There exist positive constants $c_1, c_2$ such that for any history $H$ of the implementation $I$ there is a linearization $\bar{H}$ of $H$ such that for any process $p$, if $p$ incurs $X$ RMRs in $H$ in the particular CC model under consideration and $Y$ RMRs in $\bar{H}$ in the same model, then $X \leq c_1 Y + c_2$.*

Informally, Definition 7.1 states that the RMR cost of $H$ for any process $p$ is at most a constant factor greater than the RMR cost of $\bar{H}$. Consequently, replacing a shared object with one implemented using $I$ causes at most a constant-factor increase in RMR complexity in any history. This constant factor is $c_1$. The term $c_2$ is needed to compensate for any pending operation execution by $p$ in $H$ that may be discarded from $\bar{H}$. In particular it is possible that $\bar{H}$ is empty when $H$ is not empty, in which case $Y = 0$ and $X > 0$. (In that case, $X \in O(1)$ because $H$ contains at most one operation execution by $p$ and because we assume $I$ has $O(1)$ worst-case RMR complexity).

An implementation that merely satisfies $O(1)$ RMR complexity per operation does not automatically satisfy Definition 7.1. This is because in the worst case $X$ grows linearly with the number of operations invoked on the target object in $H$, and yet $Y$ may grow much less quickly because many of these operations are applied "in-cache". In particular, in the write-back CC model if only one process is active in $H$ then $Y = O(1)$ no matter how many operations that process applies on the target object.

We now state conditions on the implementation $I$ that are sufficient for it to satisfy this RMR preservation property in the write-through and write-back CC model. We call these "locality properties". (It is straightforward to show that these locality properties imply the RMR preservation property by applying the definitions of write-through and write-back caching given in Section 2.)

For the write-through CC model, the locality property informally states that any process $p$ can apply multiple operation executions "in-cache" (i.e., on a "shared cached copy" of the object) provided these operation executions are read-like, and not interleaved with any write-like operation executions. More formally, we have:

**Definition 7.2** (locality property for the write-through CC model)**.** *Let $O_\tau$ denote the target object implemented by $I$. For any history $H$ of $I$, there is a linearization $\bar{H}$ of $H|O_\tau$ such that the following property holds:*
**(R)** *For any process $p$, if $\bar{H}'$ is a contiguous subsequence of $\bar{H}$ consisting only of read-like operation executions on $O_\tau$, and $H'$ is the subsequence of corresponding base object atomic steps in $H$, then $p$'s steps in $H'$ cause only a constant number of RMRs in $H$.*

For the write-back CC model, the locality property builds on Definition 7.2. Informally, it states that (in addition to the above) any process $p$ can apply multiple operation executions "in-cache" (i.e., on an "exclusive cached copy" of the object) provided these operation executions are not interleaved with any operation executions by other processes. More formally, we have:

**Definition 7.3** (locality property for the write-back CC model)**.** *Let $O_\tau$ denote the target object implemented by $I$. For any history $H$ of $I$, there is a linearization $\bar{H}$ of $H|O_\tau$ such that property (R) stated in Definition 7.2 holds, and furthermore the following property holds:*
**(W)** *For any process $p$, if $\bar{H}'$ is a contiguous subsequence of $\bar{H}$ consisting only of operation executions issued by $p$ on $O_\tau$, and $H'$ is the subsequence of corresponding base object atomic steps in $H$, then $p$'s steps in $H'$ cause only a constant number of RMRs in $H$.*

In the remainder of this section we present locally-accessible implementations of CAS, LL/SC, and ECAS. Note that we will use two notions of locality in our analysis—the locality of the target object and the locality of the base objects used to construct the target object. To prove that an implementation $I$ satisfies a locality property, we will first apply Definitions 7.2 and 7.3 to define our burden of proof with respect to a given history $H$ of $I$. Then, to construct such a proof, we will appeal to the RMR complexity and locality of the base objects accessed in $H$. To that end, we will consider sequences of atomic steps where some $O(1)$-RMR base object $B$ is accessed, and apply the analogs properties (R) and (W) above to those sequences. (The analog of (R) states that process $p$ incurs $O(1)$ RMRs accessing a base object $B$ in any contiguous subsequence of $H$ where no write-like operation is applied to $B$. The analog of (W) states that process $p$ incurs $O(1)$ RMRs accessing a base object $B$ in any contiguous subsequence of $H$ where only $p$ accesses $B$.)

## 7.1 Locally-Accessible CAS and LL/SC

To illustrate the techniques used for proving locality properties, we first consider the simple implementations of CAS and LL/SC from ECAS presented at the beginning of Section 6.

**Theorem 7.4.** *The implementations of CAS and LL/SC presented in Figure 13 satisfy the locality property in the DSM model and the two types of CC model under consideration, provided that the implementation of the ECAS base object $B$ satisfies the same locality property.*

*Proof.* Because in these implementations of CAS and LL/SC each access procedure applies exactly one atomic step on the base object $B$, the RMR cost of executing any access procedure equals the RMR cost of the corresponding atomic step on $B$. Now suppose that the implementation of $B$ satisfies the locality property in one of the models under consideration.
**DSM model.** The locality property follows directly from the above observation because if $B$ satisfies the locality property in the DSM model with respect to some designated process $p_{special}$, then each operation execution applied by $p_{special}$ on the target object incurs zero RMRs.
**CC model.** Recall that operation executions on the target objects are linearized at the point when the corresponding operation executions on $B$ occur, as discussed in Section 6. Now consider a history $H$ of the implementation, and let $\bar{H}$ be this particular linearization of $H|O_\tau$, where $O_\tau$ is the target object.

For property (R), fix $p$ and $\bar{H}'$ as in Definition 7.2, and consider $H'$. Since $\bar{H}'$ consists of read-like operation executions only, and since in Figure 13 a read-like operation execution on the target object only applies a read-like operation on the ECAS base object, it follows that $H'$ is a contiguous subsequence of atomic steps in $H$ that apply read-like operations on $B$. Consequently, by the locality property (R) of $B$'s implementation, $p$ incurs $O(1)$ RMRs in $H$ applying its steps from $H'$, as wanted.

The proof for property (W) is very similar. Fix $p$ and $\bar{H}'$ as in Definition 7.3, and consider $H'$. Since $\bar{H}'$ consists of operation executions by $p$ only, and since in Figure 13 an operation execution on the target object applies exactly one operation on the ECAS base object (at which point it takes effect), it follows that $H'$ is a contiguous subsequence of atomic steps in $H$ where only $p$ accesses

$B$. Consequently, by the locality property (W) of $B$'s implementation, $p$ incurs $O(1)$ RMRs in $H$ applying its steps from $H'$, as wanted. □

## 7.2 Locally-Accessible ECAS for the CC Model

The implementation of ECAS presented in Section 6 satisfies the locality property in the write-through model (Definition 7.2) when the block manager is implemented as described in Section 5. The locality property for the write-back model (Definition 7.3) does not hold, however, because a process performing $k$ successful `ECAS` operation executions in a solo history will incur $\Omega(k)$ RMRs as it accesses $k$ different blocks. (Property (W) in Definition 7.3 requires that only $O(1)$ RMRs occur in such a history.)

In the remainder of this section, we show how to modify our earlier implementation of ECAS to achieve the locality property in the write-back CC model. (As we show later, the same modifications yield the locality property in the write-through CC model.) The high-level idea is to allow a process to perform multiple write-like operation executions on an uncontended ECAS object while accessing only $O(1)$ blocks. To that end, a process $p$ executing a write-like (i.e., successful `ECAS`) operation execution will attempt to reuse the current block instead of allocating a new one. The main challenge here is to handle correctly the race condition when $p$ attempts to reuse some block $x$ while some other process $q$ attempts to access block $x$.

The modifications to the implementation from Section 6 are twofold: First, we use a $O(1)$-RMR block manager that satisfies the locality property for the write-through CC model (see Definition 7.2). (This is sufficient even in the write-back CC model.) The block manager implementation from Section 5 can be used for this purpose, as we show later on (see Lemma 7.17). Second, we override the subroutines `HelperBegin`, `HelperEnd` and `HelperCC` with the implementations shown in Figure 17. To support these subroutines, we alter the structure of a block by introducing several new fields. Boolean flags *seen*, *accessed*, and *changing* are used to keep track of whether a block has been accessed, whether an operation execution that accessed the block has taken effect, and whether the block is being reused (respectively). (In this context we refer to accesses to a block that occur outside of the block manager.) These three flags are needed for the locality property and their use is explained in the next paragraph. In addition, Boolean arrays $seenBy[1..N]$ and $accessedBy[1..N]$ are used to record which process has written *seen* and *accessed*, respectively. Statements that access these arrays are shaded to indicate that they are needed only for RMR complexity (i.e., to reduce RMRs incurred while accessing *seen* and *accessed*, as we explain later). Thus, when we reason about linearizability, these statements can be effectively ignored.

The new subroutines work as follows. Recall first that for any block $x$, the field *writer* stores the ID of the process that allocated the block, or $\perp$ for the initial block. Whenever some process $p$ applies an operation execution on an ECAS object where it accesses some block $x$, and $p \neq x \triangleright writer$, `HelperBegin(`$x$`)` announces (to the process whose ID is recorded in $x \triangleright writer$) that block $x$ has been accessed by another process by ensuring that $x \triangleright seen = \mathsf{true}$. Similarly, `HelperEnd(`$x$`)` announces that such an operation execution has taken effect by ensuring that $x \triangleright accessed = \mathsf{true}$. Function `HelperBegin(`$x$`)` also waits at line **172** for process $x \triangleright writer$ to finish reusing block $x$, if it has already started to do so. Thus, process $x \triangleright writer$ can use $x \triangleright seen$ to decide when it can reuse block $x$ safely, and can use $x \triangleright accessed$ to decide when it can perform additional RMRs without jeopardizing the locality property. Function `HelperCC(`$x$, *new*`)` allows process $x \triangleright writer$, under certain conditions, to access $x \triangleright V$ in mutual exclusion, and hence to apply a successful `ECAS` operation execution by reusing block $x$ rather than allocating a new one. The subroutines `TryToReuseBlock(`$x$`)` and `DoneReusingBlock(`$x$`)`, which are called from `HelperCC(`$x$, *new*`)`, set and reset $x \triangleright changing$ to announce that block $x$ is being reused, and should not be accessed during

that time.

The subroutines `HelperBegin(`$x$`)` and `HelperEnd(`$x$`)` use the arrays $x \triangleright seenBy[1..N]$ and $x \triangleright accessedBy[1..N]$ in addition to $x \triangleright seen$ and $x \triangleright accessed$ only to meet RMR complexity bounds imposed by locality property (R) (see Definition 7.2). To see why these arrays are needed, consider a history where processes execute only failed `ECAS` operation executions on the target object, in which case Definition 7.2 requires that each process incur $O(1)$ RMRs in the entire history. If the shaded statements that access *seenBy* and *accessedBy* were not present, processes could incur arbitrarily many RMRs writing *seen* and *accessed* in the write-through CC model. Even if the writes at line **170** and line **177** were preceded by tests checking whether *seen* and *accessed* (respectively) are already `true`, a process could incur $N - 1$ RMRs executing each test (in the write-through or write-back CC model) as every other process writes *seen* and *accessed* once. (We return to this issue in the proof of Lemma 7.14, Cases D and E.)

### 7.2.1 Analysis

Let $I_E$ denote the implementation of ECAS presented in Section 6, and let $I'_E = (\tau_{ECAS}, \mathcal{P}, \mathcal{B}, \mathcal{H})$ denote the implementation obtained by transforming $I_E$ as described earlier (i.e., replace the block manager object $M$ with one that satisfies the locality property in the write-through CC model, and implement subroutines `HelperBegin`, `HelperEnd`, and `HelperCC` as shown in Figure 17). We now establish the correctness properties of $I'_E$. The analysis follows the same approach as in Section 6.1, and so we defer the proofs of several technical lemmas (7.5–7.10) to Appendix A.3.

**Lemma 7.5.** *The analog of Lemma 6.5 for $I'_E$ holds.*

**Lemma 7.6.** *The analog of Lemma 6.6 for $I'_E$ holds.*

**Lemma 7.7.** *For any history $H$ of $I'_E$, and for any block $x$ accessed in $H$, if process $p$ allocated block $x$ (i.e., $x \triangleright writer = p$) then:*

*(a) If $p$ has completed a call to `TryToReuseBlock(`$x$`)` with response `true`, but has not subsequently made a call to `DoneReusingBlock(`$x$`)`, then no process $q \neq p$ has completed a call to `HelperBegin(`$x$`)`.*

*(b) If $p$ has completed a call to `TryToReuseBlock(`$x$`)` with response `false`, then some process $q \neq p$ has made a call to `HelperBegin(`$x$`)`. Moreover, if $p$ subsequently completes a call to `DoneReusingBlock(`$x$`)`, then some process $q' \neq p$ has made a call to `HelperEnd(`$x$`)`.*

*(c) If $p$ has completed a call to `TryToReuseBlock(`$x$`)` with response `false` during some operation execution $Op$ on the target object, then $p$ does not access block $x$ after completing $Op$.*

**Lemma 7.8.** *The analog of Lemma 6.7 for $I'_E$ holds.*

To prove linearizability, we define for any history $H \in \mathcal{H}$ a candidate linearization $\bar{H}$ as in Section 6.1, except that we augment the definition of timestamps (Definition 6.8). That is, we add a new clause (between clause (e) and clause (f)) for an `ECAS(`$isSC, cmp, new$`)` operation execution $Op$ in $H$ where line **183** is reached:

(g) Else if $p$ executes $M$.`getCurBlock()` at line **141** during $Op$, say with response $x$, and then writes $new$ to $x \triangleright V$ at line **183** of `HelperCC` during $Op$ in step $i$ of $H$, then $s = (x, i, 0)$. (If $Op$ is pending in $H$, its completion returns the value read from $x \triangleright V$ at line **143** and `true`.)

**Declarations**

**Shared variables**: (per-block)

*seenBy*[1..*N*], *accessedBy*[1..*N*] – array of Boolean, initially all false

*seen*, *accessed*, *changing* – Boolean, initially false

**Private variables**: (per-process)

*ret* – Boolean, uninitialized

**Function** `HelperBegin(`$d$`)`

**Input**: $d$ – block address

```
167 if read(d ▷ writer) ≠ PID then
168 |   if read(d ▷ seenBy[PID]) = false
    |   then
169 |   |   write d ▷ seenBy[PID] := true
170 |   |   write d ▷ seen := true
171 |   end
172 |   await d ▷ changing = false
173 end
```

**Function** `HelperEnd(`$d$`)`

**Input**: $d$ – block address

```
174 if read(d ▷ writer) ≠ PID then
175 |   if
    |   read(d ▷ accessedBy[PID]) = false
    |   then
176 |   |   write d ▷ accessedBy[PID] := true
177 |   |   write d ▷ accessed := true
178 |   end
179 end
```

**Function** `HelperCC(`$d$, *new*`)`

**Input**: $d$ – block address

**Input**: *new* – value to be written in block $d$

**Output**: Boolean success indicator (true if block $d$ was successfully reused, false otherwise)

```
180 ret := false
181 if read(d ▷ writer) = PID then
182 |   if TryToReuseBlock(d) = true then
183 |   |   write d ▷ V := new
184 |   |   write d ▷ Linked[PID] := false
185 |   |   ret := true
186 |   end
187 |   DoneReusingBlock(d)
188 end
189 return ret
```

**Function** `TryToReuseBlock(`$d$`)`

**Input**: $d$ – block address

**Output**: Boolean

```
190 write d ▷ changing := true
191 if read(d ▷ seen) = false then
192 |   return true
193 else
194 |   return false
195 end
```

**Function** `DoneReusingBlock(`$d$`)`

**Input**: $d$ – block address

```
196 write d ▷ changing := false
197 if read(d ▷ seen) = true then
198 |   await d ▷ accessed = true
199 end
```

Figure 17: Subroutines for locally-accessible ECAS implementation in CC model.

We now establish key properties of $\bar{H}$, as in Section 6.1.

**Lemma 7.9.** *The analogs of Lemma 6.10 (sequential completion) and Lemma 6.13 (order preservation) for $I'_E$ hold.*

**Lemma 7.10.** *The analog of Lemma 6.14 for $I'_E$ holds.*

**Lemma 7.11.** *The analog of Lemma 6.15 (conformity to type $\tau_{ECAS}$) for $I'_E$ holds.*

*Proof.* We modify the proof of Lemma 6.15 as follows. First, replace references to Lemmas 6.5, 6.6, 6.7 and 6.14 with references to Lemmas 7.5, 7.6, 7.8, and 7.10. Next, noting that the subroutines `HelperBegin`, `HelperEnd` and `HelperCC` do not write any of the shared variables accessed in $I_E$, except possibly the field $V$ of a block, at line **183**, we extend the case analysis as follows:
**Case F:** $Op_k$ falls under Definition 6.8 (g), and $p_k$ writes $x \triangleright V$ in some block $x$ at line **183** during the counterpart of $Op_k$ in $H$. In this case, $s_k = (s, t_k, 0)$ for some $t_k$, and $Op_k$ is an `ECAS` operation execution.

As in Case D, it follows that $p_k$ reads the value $\nu_{k-1}.V$ from $x \triangleright V$, and $\nu_{k-1}.Linked[p]$ from $x \triangleright Linked[p]$. Consequently, $Op_k$ returns $\langle \nu_{k-1}.V, \mathsf{true} \rangle$ in $\bar{H}$. As in Case D, it follows from the failure of the tests at line **144** and line **146** that $Op_k$ is successful. $S(i)$ (a) follows by the action of step $t_i$ by $p_i$, which does not change the current block but overwrites $x_i \triangleright V$ with $new_i$. $S(i)$ (b) holds since $Op_i$ returns $(\nu_{i-1}.V, \mathsf{true})$ in $\bar{H}$ and since $Op_i$ is a successful `ECAS` operation execution. □

Having established linearizability (Lemmas 7.9 and 7.11), we now consider the termination and RMR complexity.

**Lemma 7.12.** *The implementation $I'_E$ satisfies Specification 6.2 (termination).*

*Proof.* Let $H$ be a fair history of $I'_E$, and suppose for contradiction that some operation execution $Op$ on the target object does not terminate in $H$. It follows from the structure of the access procedures of $I'_E$ that $p$ makes a non-terminating call to `HelperBegin`, `HelperEnd` or `HelperCC` during $Op$. It follows from the algorithms for the subroutines under consideration that one of the following cases applies:
**Case A:** $p$ loops forever at line **172** during a call to `HelperBegin(`$x$`)`. Note that by that time, $p$ has already executed line **170** during some call to `HelperBegin(`$x$`)`, and so it follows by Lemma 7.7 (a) that any subsequent call to `TryToReuseBlock(`$x$`)` by process $w = x \triangleright writer$ returns `false`. Next, note that there is at most one such call by $w$ by Lemma 7.7 (c). This implies that $x \triangleright changing = \mathsf{false}$ holds after $w$'s last call to `TryToReuseBlock(`$x$`)` because only $w$ can assign $x \triangleright changing$ to `true`, namely at line **190**, and following each execution of this line process $w$ resets $x \triangleright changing$ at line **196** (since $H$ is fair). But this contradicts the hypothesis of Case A, which implies that $p$ repeatedly reads $x \triangleright changing = \mathsf{true}$ at line **172**.
**Case B:** $p$ loops forever at line **198** during a call to `DoneReusingBlock(`$x$`)` for some block $x$ such that $p = x \triangleright writer$. Then $p$ previously read $x \triangleright seen = \mathsf{true}$ at line **197**, and so by the algorithm some process $q$ began executing `HelperBegin(`$x$`)`. Since $p = x \triangleright writer$, it follows from $q \neq p$ and the algorithm (line **181**) that $q$ does not subsequently execute `DoneReusingBlock(`$x$`)` before calling `HelperEnd(`$x$`)`. In particular, $q$ does not access block $x$ at line **198**. Consequently, by Case A, the fairness of $H$, and the algorithms for `Read`, `LL` and `ECAS`, $q$ eventually completes a call to `HelperEnd(`$x$`)` (at line **131**, line **139** or line **164**). Moreover, during the first such call in $H$ it assigns $x \triangleright accessed = \mathsf{true}$ at line **177**. Since no process ever writes $x \triangleright accessed = \mathsf{false}$ by the algorithm, this contradicts the hypothesis of Case B. □

Next, we analyze the RMR complexity and locality properties of the implementation $I'_E$. Recall that processes incur RMRs while accessing the block manager, block allocator, and the blocks themselves. We begin by bounding the number of RMRs a process incurs in each category for each block accessed (possibly over many executions of access procedures).

**Lemma 7.13.** *For any history $H$ of $I'_E$, for any block $x$ accessed in $H$, and for any process $p$, $p$ incurs $O(1)$ RMRs in the CC model accessing the block manager and block allocator during operation executions on the target object where it accesses block $x$.*

*Proof.* First, consider the block manager object $M$. By Specification 5.1, there is at most one operation execution where $p$ allocates $x$, in which case $p$ applies a successful $M$.`chngCurBlock(...,`$x$`)` in the same operation execution. There is at most one other operation execution where $p$ accesses $x$ and calls $M$.`chngCurBlock`, namely one where $p$ calls $M$.`getCurBlock()` with response $x$ and then applies a failed $M$.`chngCurBlock(`$x$`,...)`, after which point $x$ is never again current (and hence is not accessed again by $p$) by Lemma 7.5. Thus, there are at most two operation executions in $H$ where $p$ accesses block $x$ and calls $M$.`chngCurBlock` or `AllocBlock`, each call incurring $O(1)$ RMRs. Next, consider operation executions where $p$ calls $M$.`getCurBlock()` only (and not $M$.`chngCurBlock` or `AllocBlock`). In these cases `getCurBlock` must return $x$, otherwise $p$ does not access block $x$. Process $p$ incurs $O(1)$ RMRs in $H$ accessing $M$ in such operation executions by the locality-property of $M$ (see Definition 7.2) because $M$ does not change state between $p$'s first and last `getCurBlock` in $H$ that returns $x$ by Lemma 7.5. (Recall our assumption at the beginning of this section that $M$ satisfies the locality property for the write-through CC model. We establish this property for the block manager implementation presented in Section 5 later on in Lemma 7.17.) □

**Lemma 7.14.** *For any history $H$ of $I'_E$, for any block $x$ accessed in $H$, and for any process $p$, the number of RMRs that $p$ incurs while accessing block $x$ in $H$ is:*

- *$O(1)$ in the CC model with write-back caching; and*
- *$O(1+m)$ in the CC model with write-through caching, where $m$ is the number of write-like operation executions in $H$ on the target object during which $p$ accesses block $x$.*

*Proof.* Since there are $O(1)$ fields (i.e., shared objects and subroutines) in block $x$, it suffices to show the stated upper bound on RMRs separately for each field. (In most cases, we will not distinguish between the write-through and write-back CC model, as the cost is $O(1)$ in both.)
**Case A:** variable $V$. Only one process can write $V$, namely the process $w$ that allocated block $x$. The ID of this process is stored in $x \rhd$ *writer* once $w$ completes line **153** of `ECAS` following the allocation of $x$. If $p = w$, then $p$ can only write $x \rhd V$ at line **154** (in `ECAS`, before block $x$ becomes current) and at line **183** (in `HelperCC`). On the other hand, every process may read $x \rhd V$ at line **130**, line **135** or line **143**, while executing an access procedure.
**Subcase A-i:** $p = w$, write-back model. Process $p$ holds $x \rhd V$ in exclusive mode in its cache from the moment it first writes it until some process $q \neq p$ reads it. Therefore, $p$ incurs $O(1)$ RMRs accessing $x \rhd V$ until some process $q \neq p$ reads it. By the algorithm, $q$ has completed a call to `HelperBegin(`$x$`)` by that time, and so by Lemma 7.7 (a) and the algorithm, $p$ does not write $x \rhd V$ again. As noted earlier, this implies that no process writes $x \rhd V$, and so subsequent accesses to $x \rhd V$ by any process are in-cache reads that incur $O(1)$ RMRs in total per-process. Thus, the total cost of $p$ accessing $x \rhd V$ is $O(1)$ RMRs.
**Subcase A-ii:** $p = w$, write-through model. Here $w$ incurs an RMR each time it writes $V$, which occurs at most once during an operation execution where $p$ allocates $x$, and then once per successful `ECAS` operation execution (see Case F in the proof of Lemma 7.11), which is write-like. Also, $p$

incurs $O(1)$ RMRs reading $x \triangleright V$, as in the write-back CC model (Subcase A-i). Thus, $p$ incurs $O(m)$ RMRs accessing $x \triangleright V$.

**Subcase A-iii:** $p \neq w$. Here $p$ only reads $x \triangleright V$. As argued in Subcase A-i, such reads occur only after the last write to $x \triangleright V$ (including initialization). Since $p$ does not write $x \triangleright V$ in this case, all accesses to $x \triangleright V$ by $p$, except the first, are in-cache. Thus, $p$ incurs $O(1)$ RMRs in total accessing $x \triangleright V$.

**Case B:** array $Linked[1..N]$. Note that only $p$ accesses $x \triangleright Linked[p]$, and $p$ writes this variable at most once. Thus, $p$ incurs at most two RMRs accessing this variable.

**Case C:** variable *writer*. This variable is written exactly once, by the process that allocates the block $x$. All other accesses are reads. Consequently, each process incurs at most two RMR accessing *writer*.

**Case D:** variables *seen* and $seenBy[1..N]$. The array $x \triangleright seenBy[1..N]$ is accessed similarly to $x \triangleright Linked[1..N]$, and so the analysis is analogous. Next, consider $x \triangleright seen$, which is accessed only at line **170**, line **191**, and line **197**. Because of the test at line **168**, the assignment at line **169**, and the fact that $x \triangleright seenBy[p]$ is never assigned `false`, $p$ accesses $x \triangleright seen$ at line **170** at most once. On the other hand, $p$ may access $x \triangleright seen$ multiple times at line **191** and line **197**. However, at most once such access reads `true`, since in that case `TryToReuseBlock(`$x$`)` returns `false`, and this happens at most once by Lemma 7.7 (c). Since `true` is the only value that can be written to $x \triangleright seen$, namely at line **170**, it follows that all accesses to $x \triangleright seen$ during a call to `TryToReuseBlock(`$x$`)` are in-cache, except the first and possibly the last (which returns `true`). Thus, $p$ incurs $O(1)$ RMRs in total accessing *seen* and $seenBy[1..N]$.

**Case E:** variables *accessed* and $accessedBy[1..N]$. The array $x \triangleright accessedBy[1..N]$ is accessed similarly to $x \triangleright Linked[1..N]$, and so the analysis is analogous. Next, consider $x \triangleright accessed$, which is accessed only at line **177** and line **198**. Because of the test at line **175**, the assignment at line **176**, and the fact that $x \triangleright accessedBy[p]$ is never assigned `false`, $p$ accesses $x \triangleright accessed$ at line **177** at most once. At line **198**, $p$ reads $x \triangleright accessed$ repeatedly until it reads `true`. Since $x \triangleright accessed$ is only written at line **177**, and the value written there is `true`, it follows that $p$ incurs at most two RMRs at line **198**. Thus, $p$ incurs $O(1)$ RMRs in total accessing $x \triangleright accessed$ and $x \triangleright accessedBy[1..N]$.

**Case F:** variable *changing*. Let $w$ denote the process that allocated block $x$ (i.e., $x \triangleright writer$). If $p = w$, then by the tests at line **167** and line **174**, $p$ only accesses $x \triangleright changing$ at line **190** of `TryToReuseBlock`, and at line **196** of `DoneReusingBlock`. Similarly, if $p \neq w$ then $p$ accesses $x \triangleright changing$ only at line **172** of `HelperBegin`. (This includes the case when $x$ is the initial block and $x \triangleright writer = \perp$.)

**Subcase F-i:** $p = w$, write-back model. Process $p$ holds $x \triangleright changing$ in exclusive mode in its cache from the moment it first writes it until some process $q \neq p$ reads it at line **172**. Once $q$ reaches line **172**, it follows by Lemma 7.7 (a) that $p$ does not make another call to `TryToReuseBlock(`$x$`)`, and so it executes line **190** and line **196** at most one more time with $d = x$. Thus, $p$ performs at most three RMRs accessing $x \triangleright changing$.

**Subcase F-ii:** $p = w$, write-through model. Each write of $x \triangleright changing$ at line **190** or line **196** occurs during a successful `ECAS` operation execution by $p$ in $H$, and incurs one RMR. Since there are at most two such writes per `ECAS` operation execution, $p$ incurs $O(m)$ RMRs accessing $x \triangleright changing$.

**Subcase F-iii:** $p \neq w$. After $p$ accesses $x \triangleright changing$ for the first time at line **172** of `HelperBegin`, process $w$ writes $x \triangleright changing$ at most twice more, as explained in Subcase F-i. Thus, after at most three RMRs, $p$ holds $x \triangleright changing$ in its cache and can read it locally. Since $p$ does not write $x \triangleright changing$, this implies that $p$ incurs $O(1)$ RMRs accessing $x \triangleright changing$. □

**Lemma 7.15.** *Implementation $I'_E$ satisfies the locality property in the write-through and write-back CC model (see Definitions 7.2 and 7.3).*

*Proof.* Consider any history $H$ of $I'_E$, and consider the linearization $\bar{H}$ of $H|O_\tau$ defined in our proof of conformity to type $\tau_{ECAS}$ (see Lemma 7.11), where $O_\tau$ is the target object. To prove the locality property, we will show that $p$ incurs $O(1)$ RMRs in $H$ while executing the counterparts of certain operation executions in $\bar{H}$. To that end, we will show that $p$ accesses $O(1)$ blocks during these operation executions, in which case the number of RMRs that $p$ incurs is also $O(1)$ by Lemma 7.13 and Lemma 7.14. (There can be at most one operation execution where $p$ does not access any block, namely a pending one, and it follows easily that $p$ incurs $O(1)$ in that operation execution as well.)

**Property (R)** (Definition 7.2). We must consider the write-through and write-back CC models. Fix process $p$ and a sequence $\bar{H}'$ of consecutive read-like operation executions in $\bar{H}$. Let $H'$ denote the sequence of atomic steps (which access base objects) in $H$ corresponding to $\bar{H}'$. It suffices to show that $p$ accesses at most three blocks in $H'$.

First, we will show that $p$ allocates at most one block in $H'$. Suppose, for contradiction, that $p$ allocates two or more blocks. Then this happens during the counterparts of at least two distinct `ECAS` operation executions by $p$ in $\bar{H}'$, say $Op_1$ and $Op_2$ (in that order), whose timestamps fall under Definition 6.8 (d) or (e). In fact, clause (e) must apply to both because, by our analysis in the proof of Lemma 6.15, operation executions of the other type are write-like. Consequently, by the Definition 6.8 and Definition 6.9, there is an `ECAS` operation execution $Op_e$ whose timestamp falls under Definition 6.8 (d), and which is linearized between $Op_1$ and $Op_2$ in $\bar{H}$, hence in $\bar{H}'$. Since $Op_e$ is write-like by our analysis in the proof of Lemma 6.15, this contradicts the definition of $\bar{H}'$.

Second, we will show that $M$.`getCurBlock()` returns at most two distinct values to $p$ during $H'$. Suppose, for contradiction, that $p$ receives three such values, say during the counterparts of operation executions $Op_1$, $Op_2$, and $Op_3$ (in that order) in $\bar{H}'$. Then by Definition 6.8 and Definition 6.9, there is an `ECAS` operation execution $Op_e$ in $\bar{H}$ whose counterpart in $H$ changes the current block. Furthermore, its timestamp falls under Definition 6.8 (d), and it is linearized between $Op_1$ and $Op_2$ in $\bar{H}$, hence in $\bar{H}'$. (Process $p$ may not "see" the block made current by $Op_e$ until $Op_3$, which is why we consider three operation executions by $p$.) Again, this contradicts the definition of $\bar{H}'$.

Thus, $p$ allocates at most one block and receives the addresses of at most two more from $M$.`getCurBlock` in $H'$, and so $p$ accesses at most three blocks in $H'$, as wanted.

**Property (W)** (Definition 7.3). We need only consider the write-back CC model. Fix process $p$ and a sequence $\bar{H}'$ of consecutive operation executions by $p$ in $\bar{H}$. Again let $H'$ be the corresponding sequence of atomic steps. It suffices to show that $p$ accesses at most four blocks in $H'$.

As in the proof of (R), note that if $M$.`getCurBlock()` returns three distinct values to $p$ during $H'$, then there is an `ECAS` operation execution $Op_e$ in $\bar{H}'$ whose counterpart in $H$ changes the current block. In this case, this does not lead to a contradiction, since $\bar{H}'$ may contain write-like operation executions, but it tells us that $p$ executes $Op_e$. Furthermore, we can generalize the argument easily and conclude that if $M$.`getCurBlock()` returns $k$ distinct values to $p$, then $p$ performs at least $\min(0, k-2)$ `ECAS` operation executions in $\bar{H}'$ whose counterparts in $H$ change the current block. Call this observation ($\star$).

Next, we will show that $p$ tries to change the current block at most once in $H'$. Suppose, for contradiction, that $p$ does so twice, say during the counterparts of operation executions $Op_1$ and $Op_2$ (in that order) in $\bar{H}'$. Note that the timestamps of $Op_1$ and $Op_2$ fall under Definition 6.8 (d) or (e). Suppose that $p$'s $M$.`getCurBlock()` during the counterpart of $Op_2$ in $H$ returns $x$. It follows from the algorithm and Definition 6.8 (d) that $p$ completes a call to `TryToReuseBlock(`$x$`)` during the counterpart of $Op_2$ in $H$, with response `false`, and so Lemma 7.7 (b) implies two things. First, some process $q \neq p$ has made a call to `HelperBegin(`$x$`)` by that point in $H$. Second, since $p$ completes its

subsequent call to `DoneReusingBlock(`$x$`)` during the counterpart of $Op_2$ in $H$ (otherwise it cannot call $M$.`chngCurBlock`), some process $r$ has made a call to `HelperEnd(`$x$`)` (possibly $r = q$ but not necessarily). It follows from the latter point, Definition 6.8, and Definition 6.9 that $r$'s operation execution, say $Op_r$, which also accesses $x$, appears in $\bar{H}$ and is linearized before $Op_2$.

We will now show that $Op_r$ is linearized after $Op_1$, which contradicts $\bar{H}'$ containing operation executions by $p$ only (since $Op_r$ is linearized before $Op_2$), and implies that $p$ changes the current block at most once in $H'$. Suppose otherwise, and recall that $Op_1$'s timestamp falls under Definition 6.8 (d) or (e). Assuming, without loss of generality, that $Op_1$ and $Op_2$ are the first two operation executions in $\bar{H}'$ whose counterparts in $H$ change the current block, it follows that $Op_1$'s timestamp is $(x, t_1, ...)$ for some $t_l$. (Recall that $Op_1$ and $Op_2$ fall under Definition 6.8 (d), that $p$ calls $M$.`getCurBlock()` with response $x$ during the counterpart of $Op_2$ in $H$, and that no process other than $p$ applies an operation execution between $Op_1$ and $Op_2$ in $\bar{H}'$.) At the same time, since $r$'s call to $M$.`getCurBlock()` during the counterpart of $Op_r$ in $H$ returns $x$, by Definition 6.8 $Op_r$'s timestamp is of the form $(x', t_r, ...)$ for some $t_r$, where either $x' = x$ or $x'$ is a block that becomes current after $x$. Furthermore, if $x' = x$, then $t_1 < t_r$ by Lemma 7.5 and because step $t_1$ in $H$ (which could be by $p$ or another process) makes $x$ current, whereas $r$ makes a call to $M$.`getCurBlock()` with response $x$ before step $t_r$ in $H$. In either case, $Op_r$ is linearized after $Op_1$ by Definition 6.9, as wanted.

Thus, $p$ tries to change the current block at most once in $H'$. Consequently, by our earlier observation ($\star$), $M$.`getCurBlock()` returns at most three distinct values to $p$ in $H'$. Since $p$ allocates a block only in operation executions where it tries to change the current block (by calling $M$.`chngCurBlock`), and it does so at most once per operation execution, $p$ accesses at most four blocks in total in $H'$, as wanted. □

**Theorem 7.16.** *The implementation $I'_E$ satisfies Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 6.4. Furthermore, each operation execution on the target object incurs $O(1)$ RMRs. Finally, $I'_E$ satisfies the locality property in the write-through and write-back CC models (Definitions 7.2 and 7.3).*

*Proof.* Specification 6.1 under Condition 6.4 follows directly from Lemma 7.9 and Lemma 7.11. Specification 6.2 follows directly from Lemma 7.12. $O(1)$ RMR complexity follows from Lemma 7.13, Lemma 7.14, and the fact that a process accesses at most two blocks during any operation execution on the target object. (It follows easily that an operation execution where no block is accessed also incurs $O(1)$ RMRs.) The locality property follows from Lemma 7.15. □

To complete our analysis in this section, we prove that the block manager implementation from Section 5 satisfies the locality property necessary for our purposes in this section. As stated earlier, we require only that the block manager satisfy the locality property for the write-through CC model (see Definition 7.2), even when used in the write-back CC model. (If the block manager satisfied the locality property in the write-back CC model, this would not help because every operation execution on the ECAS object that applies a write-like operation on the block manager already incurs RMRs for another reason – it calls `AllocBlock`.) To define this locality property, we classify `getCurBlock` operations as read-like, failed `chngCurBlock` operations as read-like and successful `chngCurBlock` operations as write-like.

**Lemma 7.17.** *The block manager implementation $I_{BM}$ from Section 5 satisfies the locality property stated in Definition 7.2 in both the write-through and write-back CC models provided that whenever a process $p$ calls `chngCurBlock(`$x, y$`)`, $p$'s last operation execution on the block manager was a `getCurBlock()` that returned $x$.*

*Proof.* Consider a history $H$ of the implementation $I_{BM}$ where calls to `chngCurBlock` are restricted, as stated. Let $\bar{H}$ be the linearization of $H|O_\tau$ established in Section 5, where $O_\tau$ denotes the target object. We must prove property (R), stated in Definition 7.2. Fix $p$ and $\bar{H}'$ as in Definition 7.2, and consider the sequence $H'$ of base object atomic steps corresponding to $\bar{H}'$ in $H$. We must show that $p$ incurs $O(1)$ RMRs in $H$ executing the atomic steps in $H'$. Note that no process writes $D$ in $H'$ because at that point a successful `chngCurBlock` takes effect, which is write-like, and yet we assume that $\bar{H}'$ does not contain any write-like operation executions. Consequently, each access to $D$ by $p$ in $H'$ is a read, and at most one causes an RMR. Furthermore, each such read returns the same value, say $x$, and so by the algorithm, $x \triangleright \mathit{winner}$ is not written after $p$'s first read of $D$ in $H'$, which implies that $p$ incurs at most one RMR accessing $x \triangleright \mathit{winner}$. Finally, consider pseudo-locks. It follows from the algorithm (see Figure 6) and since $p$ only reads $x$ from $D$ that the only pseudo-lock $p$ accesses in $H'$ is the one in block $x$. Process $p$ calls $x \triangleright$ `Pseudo-Enter()` and $x \triangleright$ `Pseudo-Exit()` at most once by Lemma 5.5 (a), and each call incurs $O(1)$ RMRs by Theorem 4.5. □

### 7.3 Locally-Accessible ECAS for the DSM Model

Recall that the locality property in the DSM model states that for some designated process $p_{special}$, each operation execution on the target object applied by $p_{special}$ should cost zero RMRs. The motivation behind this property is to make the implemented object behave like one supported directly in hardware, which resides entirely in one memory module and is therefore local to exactly one process. A locally-accessible implemented object may use internally multiple base objects in different memory modules, but the subset of base objects $p_{special}$ accesses must be local to $p_{special}$.

The implementation $I_E$ presented in Section 6 does not satisfy the locality property in the DSM model because $p_{special}$ may perform RMRs while accessing the block manager. This is problematic because our general approach for implementing the block manager (in Section 5) is, informally speaking, incompatible with the DSM locality property: We rely on the ability of a process executing a failed `chngCurBlock` operation execution to wait for a concurrent successful `chngCurBlock` operation execution to take effect, so that the former can be linearized after the latter. (This aspect of the implementation is handled by the pseudo-lock.) If process $p_{special}$ is the one being waited for, it cannot signal the waiting processes without performing at least one RMR if those waiting processes have all entered local-spin busy-wait loops.

In the remainder of this section we show how to modify the implementation $I_E$ of ECAS from Section 6 to achieve the locality property in the DSM model with respect to a designated process $p_{special}$. To that end, we construct an ECAS implementation $I_{E\text{-}DSM}$ by taking $I_E$ and making the base objects used therein locally-accessible to $p_{special}$. (Note that we revert to the trivial implementations of the subroutines `HelperBegin`, `HelperEnd`, and `HelperCC`, as shown in Figure 16.)

Recall that the base objects used in $I_E$ are the block manager $M$, and for each block the following: registers $V$ and *writer*, and the arrays $NextVal[1..N]$ and $Linked[1..N]$. (Since for the purposes of this paper the block allocator is not shared, it can be implemented for each process without shared objects.) We construct $I_{E\text{-}DSM}$ from $I_E$ by making all of these base objects local to $p_{special}$. Note that in the case of per-block base objects, such as $V$, this applies to *all* blocks, not just ones returned by the block allocator of $p_{special}$. This is because $p_{special}$ may access any block, even if it that block was allocated by another process. Similarly, we must make *all* elements of array $NextVal[1..N]$, and also $Linked[p_{special}]$, local to $p_{special}$.

Our task of making base objects local to $p_{special}$ is trivial for register objects, which leaves only the block manager object $M$. The implementation $I_{BM}$ described in Section 5 cannot be made local to $p_{special}$ easily for reasons described earlier. Instead, we present a new implementation that provides separate execution paths for $p_{special}$ and other processes, and is locally-accessible to $p_{special}$ in the DSM model, with RMR complexity $O(1)$ for other processes. We refer to this implementation as $I_{BM\text{-}DSM} = (\tau_{BM}, \mathcal{P}, \mathcal{B}, \mathcal{H})$, and to the "other processes" as *non-special*.

The implementation $I_{BM\text{-}DSM}$ is presented in Figure 18. This implementation is somewhat similar to $I_{BM}$ from Section 5. The address of the current block is recorded using registers $D_{special}$ and $D_{other}$. (These replace the single register $D$ in $I_{BM}$.) Each register actually stores a tuple of the form $(x, s)$, where $x$ is a block address and $s$ is an integer *sequence number*. The register containing the higher sequence number holds the address of the current block. (In case the sequence numbers are equal, $D_{other}$ determines the current block.) The access procedure for `getCurBlock()` reads $D_{special}$ and $D_{other}$ at lines **200**–**201** (using a non-atomic pair of reads), and then at lines **202**–**206** compares the sequence numbers, and returns the block address corresponding to the higher number. To apply `chngCurBlock(`$x, y$`)`, processes synchronize as follows: Non-special processes compete with each other by trying to acquire a pseudo-lock in block $x$ (line **224**). The winner of this pseudo-lock synchronizes with $p_{special}$ using a leader election algorithm (line **213** and line **225**)

that is local to $p_{special}$ (see Specification 3.9). The process $w$ that wins the LE algorithm is the one whose `chngCurBlock(`$x, y$`)` succeeds. If $w = p_{special}$, then $w$ writes $y$ and a new sequence number $nextSeq$ to $D_{special}$ (line **214**). The new sequence number is computed at lines **207**–**211**, and the algorithm for this ensures that $nextSeq$ is higher than the sequence number in $D_{other}$. This computation uses values stored in private variables $S_{special}$ and $S_{other}$, which are assigned by an earlier call to `getCurBlock()` (see Condition 7.18). If $w \neq p_{special}$, then $w$ follows similar steps; it computes $nextSeq$ at lines **226**–**230**, and then writes $D_{other}$ (line **231**). The sequence number chosen in that case is at least as high as the one in $D_{special}$.

When a process $p$ applies a failed `chngCurBlock(`$x, y$`)` (i.e., it does not win LE), it must ensure that the successful `chngCurBlock(`$x, ...$`)` by some other process $w$ has taken effect before terminating. We do this using three techniques. First, if $p \neq p_{special}$ and $w \neq p_{special}$, then $p$ waits for $w$ using a pseudo-lock (lines **224** and **242**), as in $I_{BM}$. Second, if $p = p_{special}$ and $w \neq p_{special}$, $p$ waits for $w$ using a simple busy-wait loop (lines **221** and **232**). Third, if $p \neq p_{special}$ and $w = p_{special}$, then $p$ cannot wait for $p_{special}$ because $p_{special}$ cannot signal $p$ later on without performing an RMR (as explained earlier); instead, $p$ applies a "helping mechanism" to ensure that $p_{special}$'s operation execution has taken effect. Here $p$ discovers what value $p_{special}$ is trying to write to $D_{special}$ (lines **212** and **236**), and then $p$ writes this value to $D_{special}$ (line **237**). At that point, $p_{special}$'s operation execution takes effect, unless $p_{special}$ completed its own write to $D_{special}$ at line **214** earlier. To ensure that the helping mechanism does not corrupt $D_{special}$, $p_{special}$ waits for this "round" of the mechanism to finish before applying another `chngCurBlock` (lines **217**–**219**, **234**, and **240**).

Before we begin our analysis, we introduce some useful notation for this section. When referring to the state of $D_{special}$ and $D_{other}$, or a value read from these registers, we denote by operators `Block`() and `Seq`() the block address and sequence number embedded therein. For any history $H$ of $I_{BM\text{-}DSM}$, and any state $H[i]$ that occurs in $H$, we define `MaxBlock`$(H[i])$ as `Block`$(D_{special})$ if `Seq`$(D_{special}) >$ `Seq`$(D_{other})$ in state $H[i]$, and as `Block`$(D_{other})$ otherwise. We also define `MaxSeq`$(H[i])$ as $\max($`Seq`$(D_{special}),$ `Seq`$(D_{other}))$ in state $H[i]$.

The correctness of the implementation $I_{BM\text{-}DSM}$ is contingent on the following simplifying condition, which we justify later (see Lemma 7.34):

**Condition 7.18.** *If a process $p$ calls* `chngCurBlock(`$x, y$`)`*, then:*

*(a) $p$'s last operation execution on the target object was a* `getCurBlock` *that returned $x$; and*
*(b) $p$ never before invoked* `chngCurBlock(`$x, ...$`)`*; and*
*(c) no process has invoked* `chngCurBlock(`$..., y$`)` *and $y$ is not the initial block.*

We now establish several lemmas that are used later on for proving linearizability. Proofs of lemmas 7.19–7.24 are deferred to Appendix A.4.

**Lemma 7.19.** *For any history $H$ of implementation $I_{BM\text{-}DSM}$ and for any block $x$ accessed by any process in $H$, the leader election algorithm and pseudo-lock in block $x$ are accessed according to Conditions 3.4 and 4.1.*

**Lemma 7.20.** *For any history $H$ of implementation $I_{BM\text{-}DSM}$, and any block $x$, let $H'$ be the subsequence of atomic steps in $H$ applied in executions of* `chngCurBlock(`$x, ...$`)`*. Then $H'$ either contains at most two writes to $D_{special}$ (i.e., at most one by $p_{special}$ and at most one by a non-special process) and zero writes to $D_{other}$, or it contains zero writes to $D_{special}$ and at most one write to $D_{other}$ (which is by a non-special process).*

**Lemma 7.21.** *For any history $H$ of implementation $I_{BM\text{-}DSM}$, and any block $x$, the variable $x \triangleright winner$ changes state at most once in $H$.*

**Declarations**

**Shared variables**: (global)

$D_{special}, D_{other}$ – registers, store a tuple $(b, s)$ where $b$ is a block address and $s$ is an integer, initially $(b_0, 0)$ where $b_0$ is the initial block, both local to $p_{special}$

**Shared variables**: (per-block)

$winner$ – register, stores a process ID or $\perp$, initially $\perp$, local to $p_{special}$

$helping, specialDone, helperDone$ – Boolean flags, initially false, local to $p_{special}$

$A$ – register, same type as $D_{special}$ and $D_{other}$, local to $p_{special}$

**Subroutines**: (one instance per-block)

`LeaderElect()` – $O(1)$ leader election algorithm local to $p_{special}$ (see Section 3.3)

`Pseudo-Enter()`/`Pseudo-Exit()` – $O(1)$-RMR pseudo-lock from Section 4

**Private variables**: (per-process)

$B_{special}, B_{other}$ – block addresses, uninitialized

$S_{special}, S_{other}$ – integers, uninitialized

$B$ – same type of value as $D_{special}$ and $D_{other}$

**Function** `getCurBlock()`

```
200 (B_special, S_special) := read(D_special)
201 (B_other, S_other) := read(D_other)
202 if S_other < S_special then
203 |   return B_special
204 else
205 |   return B_other
206 end
```

**Function** `chngCurBlock(`$x$, $y$`)` for $p_{special}$

```
207 if S_other < S_special then
208 |   nextSeq := S_special
209 else
210 |   nextSeq := S_other + 1
211 end
212 write x ▷ A := (y, nextSeq)
213 if x ▷ LeaderElect() = win then
214 |   write D_special := (y, nextSeq)
215 |   write x ▷ winner := p_special
216 |   write x ▷ specialDone := true
217 |   if read(x ▷ helping) = true then
218 |   |   await x ▷ helperDone = true
219 |   end
220 else
221 |   await x ▷ winner ≠ ⊥
222 end
223 return read(x ▷ winner)
```

**Function** `chngCurBlock(`$x$, $y$`)` for non-special processes

```
224 if x ▷ Pseudo-Enter() = true then
225 |   if x ▷ LeaderElect() = win then
226 |   |   if S_other < S_special then
227 |   |   |   nextSeq := S_special
228 |   |   else
229 |   |   |   nextSeq := S_other
230 |   |   end
231 |   |   write D_other := (y, nextSeq)
232 |   |   write x ▷ winner := PID
233 |   else
234 |   |   write x ▷ helping := true
235 |   |   if read(x ▷ specialDone) = false
    |   |   then
236 |   |   |   B := read(x ▷ A)
237 |   |   |   write D_special := B
238 |   |   |   write x ▷ winner := p_special
239 |   |   end
240 |   |   write x ▷ helperDone := true
241 |   end
242 |   x ▷ Pseudo-Exit()
243 end
244 return read(x ▷ winner)
```

Figure 18: Block manager implementation for DSM model.

**Lemma 7.22.** *For any history $H$ of implementation $I_{BM\text{-}DSM}$, any process $q \neq p_{special}$, and any block $x$, whenever $q$ is at lines* **236**–**237** *during a* `chngCurBlock(`$x, ...$`)` *operation execution, $p_{special}$ is continuously in a pending* `chngCurBlock(`$x, ...$`)` *operation execution where it has completed line* **212** *and not yet completed line* **219**.

**Lemma 7.23.** *For any history $H$ of implementation $I_{BM\text{-}DSM}$, and for any block $x$, if multiple* `chngCurBlock(`$x, ...$`)` *operation executions occur in $H$, then the earliest write to $D_{special}$ or $D_{other}$ that occurs during these is the only such write that changes the state of $D_{special}$ or $D_{other}$.*

**Lemma 7.24.** *For any history $H$ of implementation $I_{BM\text{-}DSM}$, any process $p$, and any step $i$ in $H$, if $p$ writes $D_{special}$ or $D_{other}$ in step $i$ of $H$, and changes the state of the variable written to $C$, then letting $y =$* `Block(`$C$`)` *and $x =$* `MaxBlock(`$H[i-1]$`)`*:*

*(a)* $y =$ `MaxBlock(`$H[i]$`)`*; and*
*(b) the sequence number in the variable written does not decrease; and*
*(c)* $|$`Seq(`$D_{special}$`)` $-$ `Seq(`$D_{other}$`)`$| \leq 1$ *in state $H[i]$; and*
*(d) the value $y$ has never been written to $D_{special}$ or $D_{other}$ before in $H$.*

We will now show that the implementation $I_{BM\text{-}DSM}$ is linearizable using the same approach as in Section 5. For each history $H$ of $I_{BM\text{-}DSM}$, we define a candidate linearization $\bar{H}$ as follows. First, for each operation execution on the target object in $H$, we assign a "timestamp" of the form $(t, q)$ where $t$ is an integer and $q$ is a process ID or 0.

**Definition 7.25.** *The timestamp $s$ for an arbitrary operation execution $Op$ in $H$, say by process $p$, and its completion (where applicable), are defined as follows:*

**Operation types** `getCurBlock()`**:**

*(a) If $Op$ is complete in $H$ and returns at line* **203**, *and $p$ reads $D_{special}$ at line* **200** *in step $i$ of $H$, then $s = (i, 0)$.*

*(b) If $Op$ is complete in $H$ and returns at line* **205**, *and $p$ reads $D_{special}$ at line* **200** *in step $i'$ of $H$, then $s = (i, p)$ where $i$ is the smallest integer $\geq i'$ such that $D_{other}$ takes on the value read by $p$ at line* **201** *in state $H[i]$.*

*(c) Otherwise $s$ is undefined, and $Op$ does not appear in $\bar{H}$.*

**Operation type** `chngCurBlock(`$x$, $y$`)`**:**

*(d) If $p = p_{special}$, and the first write to $D_{special}$ (by $p$ during $Op$ at line* **214** *or by another process at line* **237***) during a* `chngCurBlock(`$x, ...$`)` *operation execution has occurred in $H$, say in step $i$, then $s = (i, p)$.*
*(The completion of $Op$, if $Op$ is pending in $H$, returns $p$'s ID.)*

*(e) Else if $p \neq p_{special}$ and $p$ writes $D_{other}$ at line* **231** *in step $i$ of $H$, then $s = (i, 0)$.*
*(The completion of $Op$, if $Op$ is pending in $H$, returns $p$'s ID.)*

*(f) Else if $Op$ is complete, and $p$ reads $x \triangleright winner$ at line* **223** *or line* **244** *in step $i$ of $H$, then $s = (i, 0)$.*

*(g) Otherwise $s$ is undefined, and $Op$ does not appear in $\bar{H}$.*

To construct $\bar{H}$, we arrange operation executions for which timestamps are defined, in increasing order of timestamp. We prove the uniqueness of timestamps in Lemma 7.26, and define their order in Definition 7.27.

**Lemma 7.26.** *The timestamp of each operation execution in $H$ (for which the timestamp is defined) is unique.*

*Proof.* This follows because, by Definition 7.25, if the timestamp of an operation execution $Op$ by $p$ in $H$ is $(t, q)$, then either $p$ executes step $t$ in $H$ during $Op$ and $q = 0$ (clauses other than (b) and (d)), or $Op$ is pending in state $H[t]$ and $q = p$ (clauses (b) and (d)). It remains to show why $Op$ is pending in state $H[t]$ in clauses (b) and (d). For clause (b), this follows directly from Definition 7.25. For clause (d), this follows from Condition 7.18 (b) and Lemma 7.22. □

**Definition 7.27.** *For timestamps $(t_1, p_1)$ and $(t_2, p_2)$, we say that $(t_1, p_1) < (t_2, p_2)$ if and only if $t_1 < t_2$, or $t_1 = t_2$ and $p_1 < p_2$.*

**Lemma 7.28.** *For any history $H$ of implementation $I_{BM\text{-}DSM}$, and any step $i$ in $H$, if a* `getCurBlock` *operation execution $Op$ by process $p$ in $H$ has timestamp $(i, ...)$ (see Definition 7.25 (a)–(b)), and $Op$ returns $x$, then $x =$* `MaxBlock`$(H[i])$.

*Proof.* If the timestamp of $Op$ falls under Definition 7.25 (a), then $x =$ `Block`$(D_{special})$ in state $H[i]$, and $Op$ returns $x$ at line **203**. By the success of the test at line **202**, `Seq`$(D_{special})$ in state $H[i]$ is greater than `Seq`$(D_{other})$ in some later state (when $p$ reads $D_{other}$), hence in state $H[i]$ by Lemma 7.24 (b). Thus, $x =$ `MaxBlock`$(H[i])$, as wanted.

If the timestamp of $Op$ falls under Definition 7.25 (b), then $x =$ `Block`$(D_{other})$ in state $H[i]$. Suppose $p$ reads $D_{special}$ at line **200** in step $j$ during $Op$, and $D_{other}$ at line **201** in step $j' > j$ during $Op$. If $D_{other}$ does not change state between $H[j']$ and $H[j]$, then $i = j$, and $x =$ `MaxBlock`$(H[i])$ holds by the algorithm for `getCurBlock`. Otherwise, $(x, ...)$ is written to $D_{other}$ in step $i$ of $H$, and this changes the state of $D_{other}$, and so $x =$ `MaxBlock`$(H[i])$ by Lemma 7.24 (a). □

**Lemma 7.29.** *$\bar{H}$ satisfies properties (a) and (b) of linearizability (sequential completion and order preservation).*

*Proof.* Property (a) follows from our construction of $\bar{H}$ and Definition 7.25. For property (b), note that by Definition 7.25, if the timestamp of an operation execution $Op$ by $p$ in $H$ is $(i, ...)$, then $Op$ is pending in state $H[i]$ (see proof of Lemma 7.26). Thus, if $Op$ and $Op'$ are operation executions in $H$ such that $Op$ precedes $Op'$, then $Op$ has a smaller timestamp than $Op'$ by Definition 7.27, as wanted. □

It remains to prove property (c) of linearizability (conformity to type $\tau_{BM}$). To that end, we first define $Op_i$, $s_i$, $p_i$, $x_i$, $y_i$, and $\nu_i = (C_i, L_i)$ as in Section 5.1.

**Lemma 7.30.** *Implementation $I_{BM\text{-}DSM}$ satisfies property (c) of linearizability (conformity to type $\tau_{BM}$).*

*Proof.* Let $H$ be any history of $I_{BM\text{-}DSM}$. Since conformity to a type is a safety property it suffices to consider finite $\bar{H}$. Let $k = |\bar{H}|$. Define $s_0 = (b_0, 0)$ and $s_{k+1} = (\infty, 0)$, where $b_0$ is the initial block. Let $(t_i, q_i)$ denote the timestamp $s_i$.

We will prove that for any $i \in \mathbb{N}$, $0 \leq i \leq k$:

(a) For $t = t_i$ and any integer $t \in [t_i, t_{i+1})$, `MaxBlock`$(H[t]) = C_i$.

(b) If $i > 0$, then the response of $Op_i$ is the correct response for an operation execution of that type applied in state $\nu_{i-1}$.

Part (b) implies the lemma, but we require both parts for induction. Now let $S(i)$ denote parts (a)–(b) for a particular value of $i$. Note that in $H$, the state of $D_{special}$ or $D_{other}$ (hence `MaxBlock`) is changed only by an execution of line **214**, line **231**, or line **237**, which is an atomic step that defines the timestamp of an operation execution (on the target object) in $\bar{H}$. For writes to $D_{special}$ this follows from Definition 7.25 (d), Lemma 7.23, and Lemma 7.22 (which implies that if a non-special process writes $D_{special}$ in step $t$ then there is an operation execution by $p_{special}$ in $\bar{H}$ with timestamp $(t, p_{special})$). Therefore, the state of $D_{special}$ and $D_{other}$ does not change between atomic steps $t_i$ and $t_{i+1}$ in $H$. This, in turn, implies that to prove part (a) of $S(i)$, it suffices to prove that $\texttt{MaxBlock}(H[t_i]) = C_i$—and that is all we do in the inductive step that follows.

For $S(0)$, (a) follows from our earlier definition of $s_0 = (b_0, 0)$, and the initialization of $D_{special}$ and $D_{other}$ to $(b_0, 0)$. Part (b) holds trivially. Now for any $i$, $0 < i \leq k$, suppose that $S(i-1)$ holds, and consider $S(i)$. We proceed by cases on how the timestamp $s_i$ of $Op_i$ was obtained.

**Case A:** $Op_i$ falls under Definition 7.25 (a) or (b), and has timestamp $s_i = (t_i, q_i)$. In this case, $Op_i$ is a `getCurBlock` operation execution, and so $C_i = C_{i-1}$.

It follows from Lemma 7.28 that $x = \texttt{MaxBlock}(H[t_i])$, and so to prove $S(i)$ (a) and $S(i)$ (b), it suffices to show that $C_{i-1} = x$. If step $t_i$ in $H$ is a read step by $p_i$, then $x = \texttt{MaxBlock}(H[t_i])$ implies $x = \texttt{MaxBlock}(H[t_i - 1])$, and $t_{i-1} < t_i$ holds by our construction of $\bar{H}$ (Definitions 7.25 and 7.27). Thus, $S(i-1)$ (a) implies $C_{i-1} = x$, as wanted. Otherwise, step $t_i$ in $H$ is a write step by some $q \neq p_i$, and so by our construction of $\bar{H}$ (Definitions 7.25 and 7.27) the operation execution $Op'$ in $\bar{H}$ that immediately precedes $Op_i$ also has a timestamp of the form $(t_i, ...)$. Consequently, $\texttt{MaxBlock}(H[t_i]) = C_{i-1}$ holds by $S(i-1)$ part (a), and so $C_{i-1} = x$ since $\texttt{MaxBlock}(H[t_i]) = x$.

**Case B:** $Op_i$ falls under Definition 7.25 (f). In this case, $Op_i$ is a `chngCurBlock`$(x, y)$ operation execution for some $x$ and $y$, and $s_i = (t_i, 0)$ where step $t_i$ is a read of $x_i \triangleright \mathit{winner}$ by $p_i$. (We discharge this case before the remaining ones so that we can claim later on that any operation execution that falls under Definition 7.25 (f) is a failed `chngCurBlock`.)

If $p_i$ reads $x \triangleright \mathit{winner}$ at line **223** during the counterpart of $Op_i$ in $H$, then by the algorithm and Definition 7.25 (f), $p_i = p_{special}$ and $p_i$ completes line **221** before line **223**, but does not write $x \triangleright \mathit{winner}$. Since $x \triangleright \mathit{winner}$ is initially $\perp$, $p_i$ completing line **221** implies that some non-special process $p'$ wrote $x \triangleright \mathit{winner}$ at line **232** or line **238**. In fact, $p'$ must have executed line **232** otherwise by the algorithm $p$ lost $x \triangleright$ `LeaderElect()` at line **225**, and so another non-special process $p''$ won $x \triangleright$ `LeaderElect()` by Lemma 7.19 and Specification 3.5, which implies that $p'$ and $p''$ both acquired the pseudo-lock in block $x$, contradicting Lemma 7.19 and Specification 4.2 (b). (Note that $p_i$ does not win $x \triangleright$ `LeaderElect()` at line **213** by the algorithm, our assumption that $p_i$ completes line **221**, and Condition 7.18 (b).) By our construction of $\bar{H}$ (Definitions 7.25 and 7.27), the execution of line **232** by $p'$ happens during the counterpart of a `chngCurBlock`$(x, ...)$ operation execution $Op'$ that precedes $Op_i$ in $\bar{H}$. Thus, $Op_i$ is a failed `chngCurBlock`$(x, y)$ operation execution, and so $C_i = C_{i-1}$.

$S(i)$ part (a) follows from $S(i-1)$ part (a) since $C_i = C_{i-1}$ and step $t_i$ does not write $D_{special}$ or $D_{other}$.

For $S(i)$ part (b), we must show that $Op_i$ returns the ID of the process $p_j$ that applies the earliest `chngCurBlock`$(x, ...)$ in $\bar{H}$. By Lemma 7.21, since $p'$ writes a process ID to $x \triangleright \mathit{winner}$

before step $t_i$, and by the algorithm for `chngCurBlock`, $Op'$ and $Op_i$ both return $p'$. Since $Op'$ is a `chngCurBlock(`$x, ...$`)` that precedes $Op_i$ in $\bar{H}$, and its response is correct by $S(i-1)$ (b), the response of $Op_i$ is also correct.

**Case C:** $Op_i$ falls under Definition 7.25 (d). In this case, $Op_i$ is a `chngCurBlock(`$x, y$`)` operation execution for some $x$ and $y$, $s_i = (t_i, p_i)$, and $p_i = p_{special}$.

First, we will show that $Op_i$ is the earliest `chngCurBlock(`$x, ...$`)` operation execution in $\bar{H}$. (Call this observation ($\star$).) Suppose otherwise. By Definition 7.25 (d) and our analysis in Case B, the first `chngCurBlock(`$x, ...$`)` in $\bar{H}$, say $Op'$, falls under Definition 7.25 (d) or (e). Since $p_{special}$ executes $Op_i$ and we assume $Op'$ is not $Op_i$, some non-special process $p'$ executes $Op'$ by Condition 7.18 (b). Since $Op_i$ falls under Definition 7.25 (d), a write to $D_{special}$ occurs in the counterpart of some `chngCurBlock(`$x, ...$`)` in $H$, and so by Lemma 7.20 $p'$ does not write $D_{other}$ during the counterpart of $Op'$ in $H$. Since $p' \neq p_{special}$, this implies $Op'$ falls under Definition 7.25 (f), which contradicts our earlier observation that $Op'$ falls under Definition 7.25 (d) or (e). Thus, $Op_i$ is a successful `chngCurBlock(`$x, y$`)` operation execution, and so $C_i = y$.

For $S(i)$ part (a), we must show that `MaxBlock(`$H[t]$`)` $= y$. By observation ($\star$) and Definition 7.25 (d), the write to $D_{special}$ in step $t_i$ is the earliest such write during a `chngCurBlock(`$x, ...$`)`, and so by Lemma 7.23 it changes the state of $D_{special}$. Thus, it follows from Lemma 7.24 (a) that `MaxBlock(`$H[t]$`)` $= y'$ where $y'$ is the block address embedded in the value written in step $t_i$. It remains to show that $y' = y$. If $p_i$ applies step $t_i$ (at line **214**), then this occurs during the counterpart of $Op_i$ in $H$ by Condition 7.18 (b), and $y' = y$ follows directly from the algorithm for `chngCurBlock` for $p_{special}$. Otherwise, by Condition 7.18 (b), Definition 7.25 (d), and the algorithm, some non-special process $p'$ applies step $t_i$ (at line **237**) during a `chngCurBlock(`$x, ...$`)`. Furthermore, by Lemma 7.22 and lines **236**–**237**, $p'$ writes to $D_{special}$ the value $p_i$ wrote earlier to $x \triangleright A$ at line **212** during the counterpart of $Op_i$ in $H$. The block address embedded in this value is $y$, hence $y' = y$, as wanted.

For $S(i)$ part (b), we must show that $Op_i$ returns $p_i$'s ID. If the counterpart of $Op_i$ in $H$ is pending, then this follows from Definition 7.25 (d). Otherwise, $p_i$ executes lines **215** and **223** during the counterpart of $Op_i$ in $H$, and so by Lemma 7.21 and the algorithm, $Op_i$ returns $p_i$'s ID.

**Case D:** $Op_i$ falls under Definition 7.25 (e). In this case, $Op_i$ is a `chngCurBlock(`$x, y$`)` operation execution for some $x$ and $y$, and $s = (t_i, 0)$.

First, we will show that $Op_i$ is the earliest `chngCurBlock(`$x, ...$`)` operation execution in $\bar{H}$. (Call this observation ($\star$).) Suppose otherwise. As argued in Case B, the first `chngCurBlock(`$x, ...$`)` in $\bar{H}$, say $Op'$, falls under Definition 7.25 (d) or (e). In the first case, there is a write to $D_{special}$ during a `chngCurBlock(`$x, ...$`)` in $H$, which contradicts Lemma 7.20 since there is a write to $D_{other}$ during the counterpart of $Op_i$ in $H$ by Definition 7.25 (e). In the second case, there are writes to $D_{other}$ in the counterparts of $Op'$ and $Op_i$ in $H$, which again contradicts Lemma 7.20 since we assume $Op_i$ is not $Op'$. Thus, $Op_i$ is a successful `chngCurBlock(`$x, y$`)` operation execution, and so $C_i = C_{i-1}$.

For $S(i)$ part (a), we must show that `MaxBlock(`$H[t_i]$`)` $= y$, where $y$ is the second argument of $Op_i$. By Definition 7.25 (e) and the algorithm for `chngCurBlock`, the block address embedded in the value written in step $t_i$ is $y$. Moreover, this value has never been written to $D_{other}$ by the algorithm and Condition 7.18 (c), and so the write in step $t_i$ changes the state of $D_{other}$. Consequently, `MaxBlock(`$H[t_i]$`)` $= y$ follows from Lemma 7.24 (a).

For $S(i)$ part (b), we must show that $Op_i$ returns $p_i$'s ID. If the counterpart of $Op_i$ in $H$ is pending, then this follows from Definition 7.25 (e). Otherwise, $p_i$ executes lines **232** and **244** during the counterpart of $Op_i$ in $H$, and so by Lemma 7.21 and the algorithm, $Op_i$ returns $p_i$'s ID.

□

**Lemma 7.31.** *For any history $H$ of implementation $I_{BM\text{-}DSM}$, and for any* `getCurBlock` *or* `chngCurBlock` *operation execution $Op$ in $H$, say by process $p$, the number of RMRs $p$ incurs in $H$ while executing $Op$ in the DSM model is zero if $p = p_{special}$, and $O(1)$ otherwise.*

*Proof.* For $p = p_{special}$, note that $p$ incurs zero RMRs executing the leader election algorithm in each block since this algorithm is local to $p$ (i.e., satisfies Specification 3.9) by Lemma 7.19. Furthermore, any base object accessed by $p_{special}$ outside of the LE algorithm is local to $p_{special}$. For $p \neq p_{special}$, this follows from the structure of the access procedures for `getCurBlock` and `chngCurBlock`, from the RMR complexity of the leader election algorithm and pseudo-lock in each block, and from Lemma 7.19. □

**Lemma 7.32.** *The implementation $I_{BM\text{-}DSM}$ satisfies Specification 5.4 (termination).*

*Proof.* Let $H$ be any fair history of implementation $I_{BM\text{-}DSM}$. Each call to `getCurBlock` terminates in $H$ by the structure of the access procedure. Next, consider a call to `chngCurBlock` in $H$. If a leader election algorithm is executed during this call, then it terminates by Specification 3.6 and Lemma 7.19. If a pseudo-lock is accessed, then the functions `Pseudo-Enter` and `Pseudo-Exit` terminate by Lemma 7.19 and Specification 4.3, and since any process that acquires a pseudo-lock in any block $x$ (at line **224**) eventually calls $x \triangleright$ `Pseudo-Exit()` (at line **242**). Finally, we must show that any execution of the busy-wait loops at line **218** and line **221** terminates. Suppose, for contradiction, that some process loops forever in one of these.
**Case A:** line **218**. Only process $p_{special}$ may execute line **218**, and if it reaches this line, then by the success of the test at line **217**, it read $x \triangleright helping = \mathsf{true}$ earlier, and so some non-special process $p'$ wrote $x \triangleright helping = \mathsf{true}$ at line **234** by the algorithm and initialization of $x \triangleright helping$ to $\mathsf{false}$. Since $H$ is fair, $p'$ eventually writes $x \triangleright helperDone = \mathsf{true}$ at line **240**, and by the algorithm no process overwrites this variable with $\mathsf{false}$. This contradicts $p_{special}$ repeatedly reading $x \triangleright helperDone = \mathsf{false}$ at line **218**.
**Case B:** line **221**. Only process $p_{special}$ may execute line **221**, and if it reaches this line, then by the failure of the test at line **217**, if did not win $x \triangleright$ `LeaderElect()` earlier at line **213**. Consequently, by Lemma 7.19, Specification 3.5, the algorithm, and the fairness of $H$, some non-special process $p'$ wins $x \triangleright$ `LeaderElect()` at line **225**. Since $H$ is fair, $p'$ eventually writes its ID to $x \triangleright winner$ at line **232**, and by the algorithm no process overwrites this variable with $\perp$. This contradicts $p_{special}$ repeatedly reading $x \triangleright winner = \perp$ at line **221**. □

**Theorem 7.33.** *The implementation $I_{BM\text{-}DSM}$ satisfies Specifications 6.1 and 6.2. Furthermore, each operation execution on the target object incurs $O(1)$ RMRs in the DSM model. Finally, $I_{BM\text{-}DSM}$ satisfies the locality property in the DSM model with respect to the designated process $p_{special}$.*

*Proof.* Specification 6.1 (linearizability) follows directly from Lemma 7.29 and Lemma 7.30. Specification 6.2 (termination) follows directly from Lemma 7.32. $O(1)$ RMR complexity and locality follow from Lemma 7.31. □

The sequence numbers embedded in the registers $D_{special}$ and $D_{other}$ can grow without bound in $I_{BM\text{-}DSM}$. Techniques for bounding these are described in the PhD thesis of Wojciech Golab [12]. The high-level idea is that since sequence numbers are non-decreasing and `MaxSeq` grows in small increments (see Lemma 7.24 (b)–(c)), they can be represented mod $N$ and still compared correctly, with some modifications to the access procedures for `getCurBlock` and `chngCurBlock`.

To complete our analysis, we justify why Condition 7.18 holds when the implementation $I_{BM\text{-}DSM}$ is used in conjunction with the ECAS implementation from Section 6.1.

**Lemma 7.34.** *In implementation $I_E$ of ECAS from Section 6.1, the block manager base object $M$ is accessed according to Condition 7.18.*

*Proof.*
**Part (a):** This follows directly from the structure of the access procedure for operation type `ECAS`.
**Part (b):** This follows from the definition of type $\tau_{BM}$, from Lemma 6.5, and from the structure of the access procedures, whereby at most one call to `chngCurBlock`$(x, y)$ occurs, and follows a call to `getCurBlock` that returns $x$.
**Part (c):** This follows from Specification 5.1 since for each call to `chngCurBlock`$(x, y)$, the argument $y$ is obtained by first calling `AllocBlock`. □

**Theorem 7.35.** *The implementation $I_{E\text{-}DSM}$ of ECAS described in this section satisfies Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 6.4. Furthermore, each operation execution on the target object incurs $O(1)$ RMRs in the DSM model. Finally, $I_{E\text{-}DSM}$ satisfies the locality property in the DSM model with respect to the designated process $p_{special}$.*

*Proof.* Recall that $I_{E\text{-}DSM}$ is obtained from $I_E$ by

(a) declaring certain shared variables to be local to the designated process $p_{special}$, and
(b) changing the implementation of the block manager to $I_{BM\text{-}DSM}$ described in this section.

Specification 6.1 under Condition 6.4 follows from the corresponding properties of $I_E$ and $I_{BM\text{-}DSM}$ (Theorems 6.16 and 7.33). Specification 6.2 and $O(1)$ RMR complexity follow similarly. Locality of $I_{E\text{-}DSM}$ follows from the fact that, for any process $p$, (a) the shared objects accessed by $p$ other than the block manager $M$ are registers declared local to $p$; and (b) we assume that $M$ is obtained using the locally-accessible implementation $I_{BM\text{-}DSM}$ (see Theorem 7.33). □

# 8 Writable Implementations of ECAS

In this section, we describe a writable implementation of ECAS that builds on the techniques introduced in Sections 6 and 7. A *writable ECAS* is a type like ECAS except that, in addition, it supports operation type `Write` whose transition mapping is defined by (the atomic execution of) the pseudo-code shown in Figure 19. Let $\tau_{ECAS\text{-}W}$ denote this type.

```
Function Write(new)
    Input: new – value to be stored
245 V := new
246 foreach i ∈ 1..N do  Linked[i] := false
247 return OK
```

Figure 19: Definition of `Write` operation type for type $\tau_{ECAS\text{-}W}$. (The current state is denoted by $V$ and $Linked[1..N]$.)

We now present an implementation $I_{EW} = (\tau_{ECAS\text{-}W}, \mathcal{P}, \mathcal{B}, \mathcal{H})$ that is similar in spirit to the non-writable one presented in Section 6. We rely on a block manager, and define blocks so that each contains a base object $B$ of non-writable ECAS type, as well as a register *writer* that records the ID of the process that made a particular block current. The access procedures for the operation types `Read`, `LL`, `ECAS` and `Write` of $I_{EW}$ are shown in Figure 20. Lines containing shaded statements can be ignored safely for now; these statements come into play in Section 8.1 when we discuss locally-accessible implementations. For now, the subroutines `HelperBegin`, `HelperEnd` and `HelperCC` have trivial implementations (see Figure 16).

The access procedures for operation types `Read`, `LL` and `ECAS` are obtained by applying the corresponding operation to the underlying non-writable ECAS object ($B$) in the current block. The `Write(`*new*`)` operation execution attempts to change the current block to one where $B$ is initialized to *new*, by applying a `chngCurBlock(`$d, d'$`)` on the block manager at line **255**. At this point, processes applying `Write` concurrently compete to decide whose `Write` operation execution will "succeed," here meaning that it will be the last among the competing group in the linearization order. The effect of the "successful" operation execution becomes visible to subsequent operation executions, whereas the effects of the others are "overwritten" by the successful one.

One subtle point regarding the implementation that requires further explanation is the initialization of the base object $d \triangleright B$ at line **254**. A base object that is provided in hardware can be initialized by writing to it, but in this paper we assume that $d \triangleright B$ is an object implemented in software. The only implementations given so far for this object are the ones from Sections 6–7, which are not writable. However, it is straightforward to provide an initialization operation type in those implementations because of the following simplifying observations: (1) The initialization of $d \triangleright B$ is the first operation applied to this object. (2) The initialization is permitted to incur $O(1)$ RMRs because the `Write` operation execution that calls it at line **254** already incurs several RMRs. (3) At most one process at a time accesses $d \triangleright B$ at line **254**, namely the process $p$ that allocated the block $d$ (see Specification 5.1).

The procedure for initializing $d \triangleright B$ to a value *val* (which is implicit in our pseudo-code) works as follows: the calling process $p$ simply writes *val* into the field $V$ in the initial block in the implementation of $d \triangleright B$ (see Sections 6–7). (Recall that the implementation of $d \triangleright B$ internally uses blocks that are distinct from those in Figure 20.) Successive applications of this initialization procedure can be used to re-initialize $B$ – something we require later on in Section 8.1.1 (see line **277**

**Declarations**

**Shared variables**: (global)

$M$ – block manager from Section 5, initialized to the address of a fresh block

**Shared variables**: (per-block)

$B$ – instance of $O(1)$-RMR non-writable ECAS, initialized to the initial state of type $I_{EW}$ (where $V$ can be any value and $Linked[1..N] = \mathsf{false}$)

*writer* – register, stores process ID or $\perp$, initially $\perp$

**Subroutines**: (per-block)

`AllocBlock()` – block allocator from Section 5

**Private variables**: (per-process)

$d, d'$ – block addresses, uninitialized

*ret* – Boolean, initially false

**Function** `Write(`*val*`)`

```
248 d := M.getCurBlock()
249 HelperBegin(d)
250 ret := HelperCC(d, val)
251 if ret = false then
        // Try to change current block.
252     d' := AllocBlock()
253     write d' ▷ writer := PID
254     initialize d' ▷ B to a state where
        V = val and Linked[1..N] = false
255     M.chngCurBlock(d, d')
256 end
257 HelperEnd(d)
258 return OK
```

**Function** `ECAS(`*isSC, cmp, new*`)`

```
259 d := M.getCurBlock()
260 HelperBegin(d)
261 ret := (d ▷ B).ECAS(isSC, cmp, new)
262 HelperEnd(d)
263 return ret
```

**Function** `Read()`

```
264 d := M.getCurBlock()
265 HelperBegin(d)
266 ret := (d ▷ B).Read()
267 HelperEnd(d)
268 return ret
```

**Function** `LL()`

```
269 d := M.getCurBlock()
270 HelperBegin(d)
271 ret := (d ▷ B).LL()
272 HelperEnd(d)
273 return ret
```

Figure 20: Implementation $I_{EW}$ of writable ECAS.

in Figure 21 and Lemma A.11). The main caveat is that when we re-initialize $d \triangleright B$ to *val*, we must ensure not only that $V = val$ but also that $Linked[1..N] = \mathsf{false}$ in the current block underlying this object. Fortunately, this can be done using $O(1)$ RMRs because each time we re-initialize $d \triangleright B$, the current block in the implementation of $d \triangleright B$ is still the initial block of that implementation, and furthermore all elements of $Linked[1..N]$ other than possibly $Linked[p]$ are still false. (This is because we re-initialize $d \triangleright B$ only if it has been accessed by no process other than $p$.)

The proof of correctness and the RMR complexity analysis of this implementation are very similar to those of the implementation of ECAS presented in Section 6.1, and they are given in Appendix A.5. In particular, we prove:

**Theorem 8.1.** *The implementation $I_{EW}$ satisfies Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 6.4. Furthermore, each operation execution on the target object incurs $O(1)$ RMRs in the CC and DSM models.*

## 8.1 Locality

In this section we describe, for each of the shared memory models under consideration in this paper, how to transform the writable implementation $I_{EW}$ to an implementation $I'_{EW} = (\tau_{ECAS\text{-}W}, \mathcal{P}, \mathcal{B}, \mathcal{H})$ that satisfies the locality property in that model.

### 8.1.1 CC Model

In the CC model with write-through and write-back caching, we construct $I'_{EW}$ from $I_{EW}$ by making the following modifications. First, we assume that the block manager and non-writable ECAS base object $B$ (in each block) are implemented as described in Section 5 and Section 7.2, respectively. Second, we override the subroutines `HelperBegin`, `HelperEnd`, and `HelperCC` in the same way as in Section 7.2, except for a slight change in `HelperCC`: In function `HelperCC`, we replace the `Write` operation on the register $V$ (see line **183** in Figure 17) with a statement that re-initializes the base object $B$, which is the analog of $V$ here. As we show later (see Lemma A.9), only the process that allocated block $x$ will perform this for block $x$, and only before any other process has accessed $x \triangleright B$, which means that we can apply the special initialization operation discussed earlier. The modified implementation of `HelperCC` is shown in Figure 21.

In `HelperCC`, a process attempts to perform the `Write` operation execution by changing the state of $B$ in block $d$. If the process $p$ executing the `Write` is the process that allocated $d$ *and* no other process has "seen" block $d$ (i.e., the tests at lines **275** and **276** both succeed), then $p$ does not change the current block. Rather, it re-initializes $d \triangleright B$ to the argument *new* of its `Write` (line **277**); in this case, `HelperCC` returns true. Otherwise (i.e., if $p$ is not the process that allocated $d$, or some process has "seen" $d$), $p$ proceeds as before: it allocates a new block $d'$, initializes $d' \triangleright B$ to *new*, and attempts to change the current block to $d'$ (lines **252**–**255** of `Write`).

The proof of correctness and the RMR complexity analysis of this implementation are similar to those of the locally-accessible implementation of ECAS presented in Section 7.2, and they are given in Appendix A.6. In particular, we prove:

**Theorem 8.2.** *The implementation $I'_{EW}$ satisfies Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 6.4. Furthermore, each operation execution on the target object incurs $O(1)$ RMRs in the CC model. Finally, $I'_{EW}$ satisfies the locality property in the write-through and write-back CC models (Definitions 7.2 and 7.3).*

**Declarations**

**Private variables**: (per-process)

*ret* – Boolean, uninitialized

**Function** `HelperCC(`*d, val*`)`

```
274 ret := false
275 if read(d ▷ writer) = PID then
276 |   if TryToReuseBlock(d) = true then
277 |   |   re-initialize d ▷ B to a state where V = val and Linked[1..N] = false
278 |   |   ret := true
279 |   end
280 |   DoneReusingBlock(d)
281 end
282 return ret
```

Figure 21: Subroutine `HelperCC` for locally-accessible writable ECAS implementation in the CC model.

### 8.1.2 DSM Model

In the DSM model, we construct a locally-accessible ECAS implementation $I_{EW\text{-}DSM}$ from $I_{EW}$ using a transformation analogous to the one presented in Section 7.3. That is, we designate a process $p_{special}$, and make the block manager, as well as the block fields $B$ and *writer* locally-accessible to $p_{special}$. For the base object $B$, we achieve locality using the ECAS implementation from Section 7.3.

**Theorem 8.3.** *The implementation $I_{EW\text{-}DSM}$ satisfies Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 6.4. Furthermore, each operation execution on the target object incurs $O(1)$ RMRs in the DSM model. Finally, $I_{EW\text{-}DSM}$ satisfies the locality property in the DSM model with respect to the designated process $p_{special}$.*

*Proof.* This theorem is analogous to Theorem 7.35 in Section 7.3, and its proof is also analogous. As in Section 7.3, we must also show that the block manager is accessed according to Condition 7.18 in histories of $I_{EW\text{-}DSM}$, since the locally accessible block manager implementation depends on this. To that end, the analog of Lemma 7.34 follows from a proof analogous to the one given in Section 7.3 except that for Condition 7.18 (a) we consider the structure of the access procedure for operation type `Write` instead of `ECAS`. □

# 9 Simulation of Algorithms Based on Comparison Primitives and LL/SC Using Reads and Writes Only

In this Section we establish our principal result (3) from Section 1, which states: "any CC or DSM shared memory algorithm using read, write, comparison primitives and LL/SC can be simulated by an algorithm that uses only read and write operations, with only a constant-factor increase in the RMR complexity, while preserving other important properties." We explained at the end of Section 6 why our $O(1)$-RMR implementations of CAS and LL/SC from that Section are not sufficient for this purpose. (The same reasoning applies to other comparison primitives, which we discuss shortly.) Our observations there motivated the material in Sections 7 and 8. Next we will show how to simulate any comparison primitive using CAS, which we now know how to implement using reads and writes, and then finally prove our principal result (3).

## 9.1 Simulation of Comparison Primitives Using CAS

Anderson and Kim define *comparison primitives* [2] as a class of synchronization primitives that includes CAS and TAS (test-and-set). A generic comparison primitive can be thought of as an object type supporting an operation type `compare_and_fg` (in addition to `Read` and `Write`), which is parametrized by functions `f` and `g`, and corresponds to the atomic execution of the pseudo-code shown in Figure 22. As an example, this generic definition can be instantiated to CAS by defining `f(`*cmp, new*`)` $\equiv$ *new* and `g(`*old, cmp, new*`)` $\equiv$ *old*.

```
Function compare_and_fg(cmp, new)
283 old := S
284 if old = cmp then S := f(cmp,new)
285 return g(old, cmp, new)
```

Figure 22: Definition of a comparison primitive. (The current state is denoted by $S$.)

Any comparison primitive can be implemented by using a CAS implementation (e.g., the one described in Sections 6, 7 and 8 as a black box. Specifically, we record the state of the target object using a CAS object, and access the state using `CAS` operations (as well as `Read` and `Write`). To perform a `compare_and_fg` operation execution, we execute the pseudo-code above with lines **283**–**284** replaced by the following statement:

$$old := \texttt{CAS}(cmp, \texttt{f}(cmp,new))$$

Specifications 6.1 and 6.2 (linearizability and termination) follow easily from the structure of this very simple implementation. The locality property defined in Section 2 also holds if a locally-accessible CAS implementation is used, such as the one described in Sections 7 and 8.1. This follows by a straightforward proof similar to the one given for Theorem 7.4 in Section 7. (In this case we treat a `compare_and_fg` operation execution as read-like when the argument *cmp* is different from the prior state, and write-like otherwise.)

## 9.2 Principal Result

The results described up to this point culminate in Theorem 9.1 below, which states precisely our principal result (3) from Section 1. The theorem assumes that an algorithm $\mathcal{A}$ does not

access the same shared object using both a comparison primitive and LL/SC. This is a reasonable assumption because multiprocessors typically provide either one type of primitive or the other. The assumption coincides with Condition 6.4 in Section 6, which we need for technical reasons to ensure the linearizability of our implementations.

Informally, Theorem 9.1 states that any algorithm $\mathcal{A}$ can be simulated by an algorithm $\mathcal{A}'$ that does not rely on comparison primitives or LL/SC, with only a constant-factor increase in RMR complexity. Of course this claim is satisfied trivially unless the transformation preserves fundamental correctness properties. It is difficult to state precisely what properties are or are not preserved by our transformation from $\mathcal{A}$ to $\mathcal{A}'$, and so we characterize only a subset of the properties that are preserved (see properties (c)–(d) of $\mathcal{A}'$ stated in Theorem 9.1). This subset includes any safety property that can be defined (solely) in terms of the state of register objects used in $\mathcal{A}$ (which also appear in $\mathcal{A}'$), as well as common liveness properties.

As an example of how Theorem 9.1 can be applied, suppose that $\mathcal{A}$ is a mutual exclusion (ME) algorithm that uses reads, writes and LL/SC. Then $\mathcal{A}'$ is also an ME algorithm and uses reads and writes only. To see why the ME property is preserved, note that $\mathcal{A}$ (and hence $\mathcal{A}'$) can be instrumented so that any violation of ME in a history $H'$ can be inferred from $H|R$ or $H'|R$, where $R$ is the set of register objects used in $\mathcal{A}$. To that end, we introduce a global shared register $r$ that is first read and then written in each execution of the critical section. If the critical section is not executed in mutual exclusion in $\mathcal{A}'$, then in some history $H'$ of $\mathcal{A}'$ the operations on $r$ by two distinct processes inside the CS are interleaved in such a way that two consecutive operations on $r$ in $H'$ are reads. The same holds in $H$ by property (c) above, which implies that the critical section is not always executed in mutual exclusion in $\mathcal{A}$ either.

Theorem 9.1 also implies that if $\mathcal{A}$ is a ME algorithm that satisfies deadlock or starvation freedom, then $\mathcal{A}'$ satisfies the same liveness property. For starvation freedom, this follows directly from property (d) of $\mathcal{A}'$ in Theorem 9.1. For deadlock freedom, this follows from properties (c)–(d), where property (c) is used to show that if the critical section is executed finitely many times in $H'$, the same holds for $H$. (The number of times the CS is executed can be deduced by instrumenting $\mathcal{A}$ and $\mathcal{A}'$ as in the analysis of ME.)

**Theorem 9.1.** *For any of the three shared memory models discussed in this paper (i.e., write-through CC, write-back CC or DSM), and for any algorithm $\mathcal{A}$ that uses atomic read/write registers, as well as (readable/writable) shared objects that support either comparison primitives or LL/SC, let $\mathcal{A}'$ be the algorithm constructed from $\mathcal{A}$ as follows:*

1. *Simulate every application of a comparison primitives using the* `CAS` *operation type, as described in Section 9.1.*
2. *Replace any shared object accessed using* `CAS` *or* `LL/SC` *with our readable/writable, locally-accessible $O(1)$-RMR implementation. (See Section 8.1, which builds on Sections 6–8.)*

*Then $\mathcal{A}'$ has the following correctness properties:*

*(a) it uses read/write registers only; and*
*(b) it has the same RMR complexity as $\mathcal{A}$, up to a constant factor, when executed in the particular shared memory model under consideration; and*
*(c) letting $R$ denote the set of objects accessed using only reads and writes in $\mathcal{A}$, for any history $H'$ of $\mathcal{A}'$ there is a history $H$ of $\mathcal{A}$ where $H'|R = H|R$; and*
*(d) for any fair history $H'$ of $\mathcal{A}'$, there is a fair history $H$ of $\mathcal{A}$ where the same processes are active, and each active process either terminates in both histories, or does not terminate in either history.*

*Proof.*
**Property (a):** The transformation from $\mathcal{A}$ to $\mathcal{A}'$ replaces any objects other than read/write registers with our implementations, which themselves use only reads and writes, provided the base objects and subroutines are chosen as stated.
**Property (b):** Recall that the locality properties imply the RMR-preservation property (see Section 7). Step 1 in the transformation from $\mathcal{A}$ to $\mathcal{A}'$ introduces at most a constant-factor increase in RMR complexity, since an $O(1)$-RMR implementation of CAS with the locality property is used to simulate other comparison primitives. Similarly, step 2 introduces at most a constant-factor increase in RMR complexity because implementations with locality properties are used. Thus, $\mathcal{A}'$ has the same RMR complexity as $\mathcal{A}$, up to a constant factor, when both algorithms are executed on the same multiprocessor (of one of the types considered in this paper).
**Properties (c) and (d):** Suppose we are given a history $H'$ of $\mathcal{A}'$. Let $H$ be the history obtained by "reversing" the two steps in the transformation used to obtain $\mathcal{A}'$ from $\mathcal{A}$ as follows. First, we replace calls to access procedures in our implementations of CAS and LL/SC (each of which consists of one or more atomic steps) by operation executions (i.e., invocation and response pairs) on the corresponding objects of type CAS or LL/SC used in $\mathcal{A}$. Next, determine the linearization of these operation executions as defined in our proof of linearizability, and replace the operation executions with atomic steps, where each atomic step is scheduled between the corresponding invocation and response in $H'$ in a manner consistent with the linearization order. For operation executions that are pending, we record an atomic step if the corresponding operation execution on the target object has taken effect (i.e., appears in the linearization), and we discard the operation execution otherwise. Finally, we replace certain `CAS` operation executions with applications of other synchronization primitives, as needed to "reverse" step 1 of the transformation.

Since the implementations used in both steps of the transformation from $\mathcal{A}$ to $\mathcal{A}'$ satisfy Specification 6.1 (linearizability), it follows that the history $H$ is a history of $\mathcal{A}$. Furthermore, $H|R = H'|R$ follows by our construction of $H$ from $H'$, and so part (c) holds. For part (d), note that if $H$ is fair then a process is active in $H$ if and only if it is active in $H'$, and furthermore since our implementations satisfy Specification 6.2 (termination), the following property also holds: $p$ takes infinitely many steps in $H'$ only if it does so in $H$. □

Having stated Theorem 9.1 and given its proof, we now give examples of properties that are not preserved by the transformation from $\mathcal{A}$ to $\mathcal{A}'$. First, if $\mathcal{A}$ is an FCFS ME algorithm, then $\mathcal{A}'$ is not necessarily an FCFS algorithm. This is because $\mathcal{A}$ has a section of code called the *doorway*, which by definition terminates in $O(1)$ steps. The analog of this doorway also appears in $\mathcal{A}'$, but it does not necessarily terminate in $O(1)$ steps. For example, if a comparison primitive is applied in the doorway of $\mathcal{A}$, then by replacing objects that are accessed in $\mathcal{A}$ using comparison primitives or LL/SC with our simulations of these objects, we introduce busy-wait loops. (These busy-wait loops appear in pseudo-locks and the underlying name consensus algorithm.) Thus, our transformation does not in general preserve the FCFS property. Wait-freedom is another property that is not preserved, for analogous reasons.

# 10 Open Problems and Future Work

In this paper we proved three principal results. First, we showed that CAS and LL/SC are no stronger than reads and writes alone under a ranking that defines the strength of a primitive as the RMR complexity of implementing it from reads and writes only. We did so by presenting $O(1)$-RMR implementations of CAS and LL/SC from reads and writes only. Second, we strengthened our $O(1)$-RMR implementations of CAS and LL/SC with locality properties that allow these implementations to simulate their hardware counterparts not only in terms of linearizability, but also in terms of RMR complexity. We used these locally-accessible implementations to establish our third principal result, which is that any algorithm based on reads, writes, CAS and LL/SC can be transformed so that it uses reads and writes only (by replacing objects on which CAS and LL/SC are applied with our implementations) in a way that introduces at most a constant-factor increase in RMR complexity and also preserves important correctness properties.

We leave open several interesting problems related to our principal results. To begin with, in our proof of the first result we defined the synchronization primitives CAS and LL/SC as shared objects supporting certain operation types, and we assumed that a single shared object would not be accessed using both `CAS` and `LL/SC` operations. This assumption is reasonable since multiprocessors typically support either `CAS` or `LL/SC`, and not both. It is interesting to ask what would be the specification of a shared object that supports both `CAS` and `LL/SC`. We addressed this question partly by defining the ECAS type in Section 6, which provides an operation type `ECAS` that can behave either like `CAS` or like `SC`. Unfortunately, for technical reasons our implementation of ECAS is linearizable only if the `ECAS` operation is called in a consistent manner—in one history, either all calls to `ECAS` simulate `CAS`, or all calls simulate `SC`. This restriction is formalized as Condition 6.4 in our analysis. We see no reason why an $O(1)$-RMR implementation of ECAS from reads and writes without this restriction would be impossible, but we also do not know how such an implementation could be obtained.

Another open problem pertains to a potential practical application of our results. Since in practice CAS is a slower machine instruction than a read or write, it is natural to ask whether it is possible to beat the performance of a hardware CAS instruction using a software implementation of CAS based on reads and writes. As regards atomic reads and writes, it seems unlikely that this can be done in a modern multiprocessor because the performance gap between CAS and other atomic instructions is quite small. The gap is considerably larger between CAS and non-atomic reads and writes, however. We leave open the problem of how to model formally the particular non-atomic reads and writes provided by modern multiprocessors, and how such instructions can be used to simulate CAS efficiently.

We would also like to simplify our implementations conceptually. There are three main sources of complexity at hand: asynchrony, the use of weak base objects (i.e., atomic read/write registers), and our stringent RMR complexity requirements. Since asynchrony and read/write registers are inherent in the research problem, our greatest hope for reducing complexity lies in weakening the RMR complexity requirements. For example, worst-case $O(1)$ RMR complexity *per operation* is not necessary for proving our third principal result, which talks about the total number of RMRs a process performs in a history that may involve multiple operations on an implemented object. For similar reasons, our locality properties in the CC model (see Definitions 7.2 and 7.3) are stronger than necessary for proving RMR preservation (see Definition 7.1). We leave open the question of how the RMR preservation property can be achieved using weaker RMR complexity per-operation and locality properties. Another interesting possibility, suggested by Sam Toueg, is to relax the RMR preservation property itself by looking at the total number of RMRs performed in an entire history by *all* processes rather than by each process individually.

**Acknowledgment.** Sincere thanks to Angela Demke Brown, Sam Toueg, Prasad Jayanti and Alan Borodin for their insightful comments on this work while serving on the PhD committee of Wojciech Golab. We would also like to thank Faith Ellen for her feedback on presentations of this work. We are also deeply indebted to the anonymous referees for their helpful suggestions and careful reading of this work, including the shorter conference version.

## References

[1] J. Anderson and Y.-J. Kim. A generic local-spin fetch-and-$\phi$-based mutual exclusion algorithm. *J. of Parallel Distributed Computing*, 67(5):551–580, 2007.

[2] J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. *Distributed Computing*, 15(4):221–253, 2002.

[3] J. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16:75–110, 2003.

[4] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[5] H. Attiya, D. Hendler, and P. Woelfel. Tight RMR Lower Bounds for Mutual Exclusion and Other Problems. In *Proc. of 40th STOC*, 2008.

[6] T. Craig. Queuing spin lock algorithms to support timing predictability. In *Proc. of 14th RTSS*, pages 148–156, 1993.

[7] R. Cypher. The communication requirements of mutual exclusion. In *Proc. of 7th SPAA*, pages 147–156, 1995.

[8] R. Danek and W. Golab. Closing the Complexity Gap Between FCFS Mutual Exclusion and Mutual Exclusion. In *Proc. of 22nd DISC*, pages 93–108, 2008.

[9] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[10] C. Dwork, M. Herlihy and O. Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, 1997.

[11] R. Fan and N. Lynch. An $\Omega(n \log n)$ lower bound on the cost of mutual exclusion. In *Proc. of 25th PODC*, pages 275–284, 2006.

[12] W. Golab. Constant-RMR Implementations of CAS and Other Synchronization Primitives Using Read and Write Operations. Ph.D. Thesis, University of Toronto, 2010.

[13] W. Golab, D. Hendler, and P. Woelfel. An O(1) RMRs leader election algorithm. *SIAM Journal on Computing*, 39(7): 2726–2760, 2010.

[14] P. Jayanti. A Complete and Constant Time Wait-Free Implementation of CAS from LL/SC and Vice Versa. In *Proc. of 12th DISC*, pages 216–230, 1998.

[15] P. Jayanti and S. Toueg. Some Results on the Impossibility, Universality, and Decidability of Consensus. In *Proc. of 6th WDAG*, pages 69–84, 1992.

[16] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[17] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[18] Y.-J. Kim and J. Anderson. Adaptive Mutual Exclusion with Local Spinning. *Distributed Computing*, 19(3):197-236, 2007.

[19] Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proc. 15th DISC*, pages 1–15, London, UK, 2001.

[20] V. Luchangco, M. Moir, and N. Shavit. On the Uncontended Complexity of Consensus. In *Proc. 17th DISC*, pages 45–59, 2003.

[21] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[22] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proc. of 16th PODC*, pages 219–228, 1997.

[23] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, San Francisco, California, 1994.

[24] J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995.

# A Appendix

## A.1 Analysis from Section 4

*Proof of Theorem 4.5.* First, note that when Condition 4.1 holds, each process calls `Pseudo-Enter()` at most once, and so the name consensus algorithm is accessed according to Condition 3.1 (i.e., at most once by each process). Thus, we can appeal to Specifications 3.2 and 3.3 (of name consensus) in our analysis below.

**Specification 4.2:** Property (a) follows easily from the algorithm. Property (b) holds because `Pseudo-Enter()` returns `true` only if the caller won `NameDecide()` at line **50**, which happens for at most one process by Specification 3.2. Property (c) holds because `Pseudo-Enter()` returns `false` only after the caller reads *flag* = `true` at line **54**, which does not happen until the process that won name consensus completes line **58** of `Pseudo-Exit()`. This process is the one whose `Pseudo-Enter()` returned `true`, and which has called `Pseudo-Exit()` by the time *flag* = `true` holds.

**Specification 4.3:** Let $H$ by any fair history where Condition 4.1 holds. For any call to function `Pseudo-Enter()` in $H$, `NameDecide()` at line **50** terminates by Specification 3.3. The process that won `NameDecide()` (which is unique by Specification 3.2) then completes `Pseudo-Enter()` after $O(1)$ additional steps, which implies property (a). Since the busy-wait loop at line **54** of `Pseudo-Enter()` terminates after the first write of *flag* at line **58** of `Pseudo-Exit()`, property (b) also holds. Finally, property (c) follows directly from the structure of `Pseudo-Exit()`.

**RMR complexity:** This follows from the structure of `Pseudo-Enter()` and `Pseudo-Exit()`, from the RMR complexity of the name consensus algorithm, and from the fact that the busy-wait loop at line **54** terminates after at most two RMRs (one at the first access to *flag*, and possibly one more once *flag* is overwritten with `true` at line **58**). □

## A.2 Analysis from Section 5

*Proof of Lemma 5.5.*

**Part (a):** Since a process calls `Pseudo-Exit` only at line **92**, and only after calling `Pseudo-Enter` at line **89**, it suffices to show that $p$ accesses the pseudo-lock in block $x$ at most once. Suppose, for contradiction, that $H$ is the shortest history at the end of which $p$ is about to call $x \triangleright$ `Pseudo-Enter()` for the second time. The first time $p$ calls $x \triangleright$ `Pseudo-Enter()` in $H$, this happens at line **89**, and before $p$ reaches line **95** in the corresponding call to `chngCurBlock`, some process assigns a value different from $\perp$ to $x \triangleright$ *winner* at line **91**. This is because either $p$ does so, or by Specification 4.2 (c) (and minimality of $|H|$) some other process does so and then calls $x \triangleright$ `Pseudo-Exit()` at line **92** before $p$ completes its call to $x \triangleright$ `Pseudo-Enter()`. Thus, if $p$ subsequently accesses block $x$ inside `chngCurBlock`, then $x \triangleright$ *winner* $\neq \perp$ holds at line **88**, and so $p$ does not access the pseudo-lock in block $x$ at all. This contradicts the hypothesis that $p$ is about to call $x \triangleright$ `Pseudo-Enter()` for the second time at the end of $H$.

**Part (b):** Consider the value a process reads from $x \triangleright$ *winner* at line **95** of `chngCurBlock`. Since we showed above that the pseudo-lock in block $x$ is accessed according to Condition 4.1, it follows from the algorithm (Figure 9) that at most one process writes $x \triangleright$ *winner*, namely the winner of the pseudo-lock in block $x$. (We appeal to Specification 5.1 (b) implicitly here and in many other proofs.) We also argued above that by the time a process reaches line **95**, some process has executed line **91**, hence line **90**. These two observations imply the lemma. □

## A.3 Analysis from Section 7.2

*Proof of Lemma 7.5.* This follows by the same proof as given in Section 6.1. □

*Proof of Lemma 7.6.* This follows by the same proof as given in Section 6.1, with any reference to Lemma 6.5 replaced by a reference to Lemma 7.5. □

*Proof of Lemma 7.7.*
**Part (a):** Note that $x \triangleright \mathit{changing} = \mathsf{true}$ holds from the moment $p$ completes line **190** during its last call to `TryToReuseBlock(`$x$`)` until the point in $H$ under consideration. Consequently, if $q$ completed a call to `HelperBegin(`$x$`)`, then it must have completed line **172** before $p$ last executed line **190**. But in that case $q$'s execution of lines **168**–**171** precedes $p$'s last execution of line **191**. It then follows by the algorithm that $x \triangleright \mathit{seen} = \mathsf{true}$ holds when $p$'s executes line **191**. This contradicts $p$ executing line **192**.
**Part (b):** If $p$ has completed a call to `TryToReuseBlock(`$x$`)` with response false then it read $d \triangleright \mathit{seen} = \mathsf{true}$ at line **191**, which implies that some process $q \neq p$ previously executed line **170**, and hence $q$ made a call to `HelperBegin(`$x$`)`. Since $x \triangleright \mathit{seen} = \mathsf{true}$ is a stable property, if $p$ subsequently completes a call to `DoneReusingBlock(`$x$`)` then it executes line **198**, and does not complete that line until some process $q' \neq p$ executes line **177** of `HelperEnd(`$x$`)`.
**Part (c):** If `TryToReuseBlock(`$x$`)` returns false, then the calling function `HelperCC` also returns false. Furthermore, $p$ executes $M$`.chngCurBlock(`$x$`, `$y$`)` at line **156** (see Figure 15) for some block $y$ during its `ECAS` operation execution under consideration before it executes another operation execution on the target object. Once this `chngCurBlock` occurs, $x$ is no longer the current block by Lemma 7.5. Consequently, once $p$ completes $Op$, it never accesses block $x$ again. □

*Proof of Lemma 7.8.* The proof of Lemma 6.7 given in Section 6.1 breaks, even after replacing references to Lemma 6.5 by references to Lemma 7.5. This is because inside function `HelperCC(`$d$`, `$\mathit{new}$`)`, a process may write $d \triangleright V$ at line **183** after $d$ has become current. To fix the proof, we must show that for any block $x$, once some process $q$ has read $x \triangleright V$ at line **143**, no process $p \neq q$ overwrites $x \triangleright V$ at line **183**. Suppose otherwise, and note that $p = x \triangleright \mathit{writer}$ by the test at line **181**. Furthermore, when $p$ is at line **183**, $q$ has completed a call to `HelperBegin(`$x$`)`, $p$ has completed a call to `TryToReuseBlock(`$x$`)` with response true (at line **182**), and $p$ has not subsequently called `DoneReusingBlock(`$x$`)` (at line **187**). But this contradicts Lemma 7.7 (a). □

*Proof of Lemma 7.9.* Both lemmas follow for $I'_E$ by the same proofs as in Section 6.1, with reference to Lemmas 6.5 and 6.6 replaced by references to Lemmas 7.5 and 7.6. □

*Proof of Lemma 7.10.* We modify the proof of Lemma 6.14 given in Section 6.1 as follows. First, we replace references to Lemma 6.5 by references to Lemma 7.5. Second, we must consider the case when $Op_e$ or $Op_i$ is an `ECAS` operation execution whose timestamp falls under clause (g) above.

If $Op_i$ falls under clause (g) and $Op_e$ does not, then we deal with $Op_i$ in the same way as when $Op_i$ is a `Read` or `LL`. (See Case A in the proof of Lemma 6.14.)

If $Op_e$ falls under clause (g), then it must be that the same process (i.e., $p$) applies $Op_l$ and $Op_e$. To see this, note that since $Op_e$ is the first successful `ECAS` that occurs between $Op_l$ and $Op_i$ in $\bar{H}$, it follows from Definitions 6.8 and 6.9 (as augmented in this section) that the $M$`.getCurBlock()` in the counterparts of both $Op_l$ and $Op_e$ in $H$ returns $x$. Furthermore, since $Op_l$ occurs before $Op_e$ in $\bar{H}$, the call to `HelperBegin(`$x$`)` during the counterpart of $Op_l$ in $H$ occurs before the call to `DoneReusingBlock(`$x$`)` during the counterpart of $Op_e$ in $H$, and so Lemma 7.7 part (a) implies that the same process executes $Op_l$ and $Op_e$. Thus, $p$ applies $Op_l$, and then $Op_e$, which assigns $x \triangleright \mathit{Linked}[p] = \mathsf{false}$ at line **184**. Since $p$ does not apply an `LL` between $Op_e$ and $Op_i$, it follows from the algorithm that $p$ reads $x \triangleright \mathit{Linked}[p] = \mathsf{false}$ during the counterpart of $Op_i$ in $H$, which contradicts the definition of $Op_i$. □

## A.4 Analysis from Section 7.3

*Proof of Lemma 7.19.* This follows immediately from Condition 7.18 (b) and the structure of the access procedures. □

*Proof of Lemma 7.20.* A write to $D_{special}$ can only occur at line **214** or line **237**, and a write to $D_{other}$ can only occur at line **231**. By Condition 7.18 (b) and the algorithm, $H'$ contains at most one write to $D_{special}$ or $D_{other}$ by $p_{special}$, and this is a write to $D_{special}$. Similarly, for any non-special process, $H'$ contains at most one write to $D_{special}$ or $D_{other}$. Furthermore, any non-special process that applies such a write must first acquire the pseudo-lock in block $x$ at line **224**, and so there is at most one such process by Lemma 7.19 and Specification 4.2 (b). Thus, $H'$ contains at most one write to $D_{special}$ or $D_{other}$ by any non-special process.

To complete the proof, it suffices to rule out the case when $H'$ contains writes to both $D_{special}$ and $D_{other}$. As explained earlier, it follows in this case that a non-special process $q$ writes $D_{other}$ and that $p_{special}$ writes $D_{special}$. Furthermore, by the algorithm $p_{special}$ wins $x \triangleright$ `LeaderElect()` at line **213** before applying its write at line **214**, and $q$ wins $x \triangleright$ `LeaderElect()` at line **225** before applying its write at line **231**. Thus, $p_{special}$ and a non-special process both win $x \triangleright$ `LeaderElect()` in $H$, which contradicts Lemma 7.19 and Specification 3.5. □

*Proof of Lemma 7.21.* Two writes to $x \triangleright$ *winner* that assign different values can occur only at line **215** and line **232**, or at line **232** and line **238**. In the first case, a write to $D_{special}$ (at line **214**) occurs during some `chngCurBlock`$(x, ...)$ operation execution, and a write to $D_{other}$ (at line **231**) also occurs during some `chngCurBlock`$(x, ...)$, which contradicts Lemma 7.20. In the second case, a write to $D_{other}$ (at line **231**) occurs during some `chngCurBlock`$(x, ...)$ operation execution, and a write to $D_{special}$ (at line **237**) also occurs during some `chngCurBlock`$(x, ...)$, which again contradicts Lemma 7.20. □

*Proof of Lemma 7.22.* Let $Op_q$ denote $q$'s `chngCurBlock`$(x, ...)$ operation execution under consideration and note that by Condition 7.18 (b) there is at most one such operation execution in $H$.

First, we will prove that when $q$ reaches line **236** during $Op_q$, process $p_{special}$ has already completed line **212** during its own `chngCurBlock`$(x, ...)$ operation execution. Note that before reaching line **236**, $q$ acquires the pseudo-lock in block $x$ at line **224**, and then loses $x \triangleright$ `LeaderElect()` at line **225**. Since $q$ loses $x \triangleright$ `LeaderElect()`, by Specification 3.5 another process must make a call to $x \triangleright$ `LeaderElect()` before $q$ completes its call. By Lemma 7.19, Specification 4.2 (b) and the algorithm, the only other process that may call $x \triangleright$ `LeaderElect()` is $p_{special}$ at line **213** during a `chngCurBlock`$(x, ...)$ operation execution. Now let $Op_{special}$ denote this `chngCurBlock`$(x, ...)$ operation execution by $p_{special}$ and note that it is unique in $H$ by Condition 7.18 (b). Since $p_{special}$ makes its call to $x \triangleright$ `LeaderElect()` at line **213** before $q$ completes its own call at line **225**, $p_{special}$ completes line **212** during $Op_{special}$ before $q$ reaches line **236** during $Op_q$, as wanted.

To complete the proof, we will show that $q$ completes line **237** during $Op_q$ before $p_{special}$ completes line **219** during $Op_{special}$. If $q$ reaches line **237** at all during $Op_q$, it does so after reading $x \triangleright \mathit{specialDone} =$ `false` at line **235**, which by the algorithm occurs before $p_{special}$ writes $x \triangleright \mathit{specialDone} =$ `true` at line **216**. Thus, $p_{special}$ reads $x \triangleright \mathit{helping} =$ `true` at line **217** after $q$ writes $x \triangleright \mathit{helping} =$ `true` at line **234**, and so $p_{special}$ branches to line **218** during $Op_{special}$. Before $p_{special}$ completes this line, some process $z$ writes $x \triangleright \mathit{helperDone} =$ `true`, at line **240** during a `chngCurBlock`$(x, ...)$ after acquiring the pseudo-lock in block $x$ at line **224**. Since $q$ wins this pseudo-lock before reaching line **219**, it follows from Lemma 7.19 and Specification 4.2 (b) that $z = q$. Since $Op_q$ is $q$'s only `chngCurBlock`$(x, ...)$ in $H$, as explained earlier, this implies that $q$

completes line **240** during $Op_q$ before $p_{special}$ completes line **218** during $Op_{special}$, hence $q$ completes line **237** during $Op_q$ before $p_{special}$ completes line **219** during $Op_{special}$, as wanted. □

*Proof of Lemma 7.23.* If multiple writes to $D_{special}$ or $D_{other}$ occur in `chngCurBlock(`$x, ...$`)` operation executions, then by Lemma 7.20 there are exactly two of these, one by $p_{special}$ and one by some non-special process $q$. Furthermore, both of these writes apply to $D_{special}$.

To prove the lemma, we first show that $q$'s write at line **237** assigns the same value as the write by $p_{special}$ at line **214** during the two `chngCurBlock(`$x, ...$`)` operation executions under consideration. To that end, note that by Lemma 7.22, $q$'s read of $x \rhd A$ at line **236** occurs after $p_{special}$'s write of $x \rhd A$ at line **212** during these operation executions. Since $x \rhd A$ is written at most once in $H$ by the algorithm and Condition 7.18 (b), the value $p_{special}$ writes to $x \rhd A$ at line **212** is the same as the value $q$ reads from $x \rhd A$ at line **236**, hence the same as the value $q$ writes to $D_{special}$ at line **237**.

Finally, we must prove that no process $z$ writes $D_{special}$ between the two writes under consideration by $p_{special}$ and $q$, regardless of the order in which they occur. First, note that by Lemma 7.22 and the algorithm, $p_{special}$ is continuously in a `chngCurBlock(`$x, ...$`)` operation execution between these two writes. Since $p_{special}$ writes $D_{special}$ at most once in each `chngCurBlock`, this implies $z \neq p_{special}$. Furthermore, if $z \neq p_{special}$ then by Lemma 7.22 $z$'s write to $D_{special}$ must occur during a `chngCurBlock(`$x, ...$`)` by $z$, which implies that $D_{special}$ is written three times (i.e., by $p_{special}$, $q$ and $z$) in a `chngCurBlock(`$x, ...$`)`, contradicting Lemma 7.20. □

*Proof of Lemma 7.24.* Since the properties stated are safety properties, it suffices to consider finite $H$. Let $S(j)$ denote parts (a)–(d) for a history $H$ of length $j$. We proceed by induction on $j$. For $S(0)$, all parts follow trivially. Now for any $j > 0$, suppose that $S(j-1)$ holds, and consider $S(j)$. It suffices to consider the case when $p$ writes $D_{special}$ or $D_{other}$ in step $j$ of $H$, and changes the state of the variable written to $C$. It follows from the algorithm that such a step must occur during some `chngCurBlock(`$x', y'$`)` operation execution $Op$ by $p$, for some $x'$ and $y'$.

**Parts (a)–(c):** Suppose that $x$ is written to $D_{special}$ or $D_{other}$ for the first time in step $j'$ (where $j' = 0$ if $x$ is the initial block $b_0$). It follows from $S(j-1)$ (a) and the initialization of $D_{special}$ and $D_{other}$ that $x =$ `MaxBlock(`$H[j']$`)`. Since $x =$ `MaxBlock(`$H[j-1]$`)`, it follows from $S(j-1)$ parts (a), (b) and (d) that neither $D_{special}$ nor $D_{other}$ changes state in $H$ after step $j'$ and before step $j$. Call this observation ($\star$).

Now consider the sequence number `Seq(`$C$`)`. This number is computed and recorded in private variable $nextSeq$, either at lines **207**–**211** or lines **226**–**230**, using as inputs the values $S_{special}$ and $S_{other}$ recorded at lines **200**–**201** during a prior call to `getCurBlock` that returns $x$. The latter call is either by $p$ (see Condition 7.18 (b)), or by $p_{special}$ if $p$ obtains the new sequence number from $x \rhd A$ at line **236** (see Lemma 7.22 and lines **207**–**212**). In particular, during the computation of $nextSeq$, $S_{special}$ and $S_{other}$ are the sequence numbers read from $D_{special}$ and $D_{other}$ at lines **200**–**201**, respectively, during a `getCurBlock` that returns $x$. Now by our choice of $j'$, $x$ was read from either $D_{special}$ or $D_{other}$ after step $j'$, and so by $S(j-1)$ (b) and $x =$ `MaxBlock(`$H[j']$`)` it follows that $\max(S_{special}, S_{other}) \geq$ `MaxSeq(`$H[j']$`)`. At the same time, by $S(j-1)$ (b) and observation ($\star$), it follows that $\max(S_{special}, S_{other}) \leq$ `MaxSeq(`$H[j']$`)`. Thus, $\max(S_{special}, S_{other}) =$ `MaxSeq(`$H[j']$`)`, and so by the computation of `Seq(`$C$`)` at lines **207**–**211** or lines **226**–**230**, either `Seq(`$C$`)` = `MaxSeq(`$H[j']$`)` or `Seq(`$C$`)` = `MaxSeq(`$H[j']$`)` $+ 1$. $S(j)$ (b) follows from this and from observation ($\star$). It remains to prove $S(j)$ (a) and $S(j)$ (c).

**Case A:** $p = p_{special}$ and $p$'s write in step $j$ occurs at line **214**. Consider $p$'s prior computation of $nextSeq$ at lines **207**–**211**.

If $S_{other} < S_{special}$ then $p$ read $x$ from $D_{special}$ during its last `getCurBlock`, and $nextSeq =$

$S_{special}$. By observation ($\star$) and our definition of $j'$, $D_{special}$ still contains $(x, S_{special})$ in state $H[j-1]$, $D_{other}$ does not contain $(x, ...)$ in state $H[j-1]$, and $x = \texttt{MaxBlock}(H[j-1])$. Thus, when $p$ writes $(y, S_{special})$ to $D_{special}$ in step $j$, $\texttt{MaxSeq}(H[j]) = nextSeq$ and $y = \texttt{MaxBlock}(H[j])$, which implies $S(j)$ (a). Since $\texttt{Seq}(D_{special})$ does not change, $S(j)$ (c) follows from $S(j-1)$ (c).

If $S_{other} \geq S_{special}$ then $p$ read $x$ from $D_{other}$ during its last `getCurBlock`, and $nextSeq = S_{other} + 1$. By observation ($\star$) and our definition of $j'$, $D_{other}$ still contains $(x, S_{other})$ in state $H[j]$, $D_{special}$ does not contain $(x, ...)$ in state $H[j-1]$, and $x = \texttt{MaxBlock}(H[j-1])$. Thus, when $p$ writes $(y, S_{other}+1)$ to $D_{special}$ in step $j$, $\texttt{MaxSeq}(H[j]) = nextSeq$ and $y = \texttt{MaxBlock}(H[j])$, which implies $S(j)$ (a). Since $\texttt{Seq}(D_{special}) = \texttt{Seq}(D_{other}) + 1$ in state $H[j]$, $S(j)$ (c) also follows.

**Case B:** $p \neq p_{special}$ and $p$'s write in step $j$ occurs at line **237** of `chngCurBlock`. As explained earlier, the value $p$ writes is based the computation of $nextSeq$ by $p_{special}$ at lines **207**–**211**. Thus, when $p$ writes $D_{special}$ in step $j$, $S(j)$ (a) and $S(j)$ (c) follow as in Case A.

**Case C:** $p \neq p_{special}$ and $p$'s write in step $j$ occurs at line **231**. As in Case A, consider $p$'s prior computation of $nextSeq$ at lines **226**–**230**.

If $S_{other} < S_{special}$ then $p$ read $x$ from $D_{special}$ during its last `getCurBlock`, and $nextSeq = S_{special}$. By observation ($\star$) and our definition of $j'$, $D_{special}$ still contains $(x, S_{special})$ in state $H[j]$, $D_{other}$ does not contain $(x, ...)$ in state $H[j-1]$, and $x = \texttt{MaxBlock}(H[j-1])$. Thus, when $p$ writes $(y, S_{special})$ to $D_{other}$ in step $j$, $\texttt{MaxSeq}(H[j]) = nextSeq$ and $y = \texttt{MaxBlock}(H[j])$, which implies $S(j)$ (a). Since $\texttt{Seq}(D_{special}) = \texttt{Seq}(D_{other})$ in state $H[j]$, $S(j)$ (c) also follows.

If $S_{other} \geq S_{special}$ then $p$ read $x$ from $D_{other}$ during its last `getCurBlock`, and $nextSeq = S_{other}$. By observation ($\star$) and our definition of $j'$, $D_{other}$ still contains $(x, S_{other})$ in state $H[j]$, $D_{special}$ does not contain $(x, ...)$ in state $H[j-1]$, and $x = \texttt{MaxBlock}(H[j-1])$. Thus, when $p$ writes $(y, S_{other})$ to $D_{other}$ in step $j$, $\texttt{MaxSeq}(H[j]) = nextSeq$ and $y = \texttt{MaxBlock}(H[j])$, which implies $S(j)$ (a). Since $\texttt{Seq}(D_{special}) = \texttt{Seq}(D_{other})$ in state $H[j]$, $S(j)$ (c) also follows.

**Part (d):** First, we will show that $y$ is the second argument of some `chngCurBlock` operation execution in $H$. This is either $p$'s operation execution $Op$ under consideration, if step $j$ occurs at line **214** or line **231**, or by Lemma 7.22 and the algorithm it is a `chngCurBlock(`$x, ...$`)` operation execution $Op'$ by $p_{special}$ (that $Op$ is "helping") if step $j$ occurs at line **237**. Furthermore, in the latter case, by Lemma 7.23 $p_{special}$ does not change the state of $D_{special}$ or $D_{other}$ during $Op'$ (because $p$ does during $Op$). Thus, there is a distinct `chngCurBlock(...,`$y$`)` operation execution in $H$ for every step in $H$ that changes the state of $D_{special}$ or $D_{other}$ to $(y, s)$ for some sequence number $s$. This and Condition 7.18 (c) imply part (d) since by Condition 7.18 (c), $y$ is unique and different from the initial block in each such operation execution. □

## A.5 Analysis from Section 8

**Lemma A.1.** *The analog of Lemma 6.5 for $I_{EW}$ holds.*

*Proof.* This follows by a proof analogous to the one given in Section 6.1. References to line **152** and line **156** are replaced by references to line **252** and line **255**. □

**Lemma A.2.** *The analog of Lemma 6.6 for $I_{EW}$ holds.*

*Proof.* This follows by the same proof as given in Section 6.1, with any reference to Lemma 6.5 replaced by a reference to Lemma A.1. □

Now consider linearizability. For any history $H$ of the implementation, we define a candidate linearization $\bar{H}$ using the same general approach for (non-writable) ECAS in Section 6.1.1.

**Definition A.3.** *The timestamp $s$ for an arbitrary operation execution $Op$ in $H$, say by process $p$, and its completion (where applicable), are defined as follows:*

**Operation type `ECAS`:** *(and similarly for `Read` and `LL`, which are implemented in an analogous way)*

(a) *If $p$ executes $M$.`getCurBlock()` at line* **259** *during $Op$, say with response $x$, and accesses $x \triangleright B$ at line* **261** *in step $i$ of $H$, then $s = (x, i, 0)$.*
*(If $Op$ is pending in $H$, its completion returns the value returned by the base object atomic step on $x \triangleright B$ in step $i$.)*

(b) *Otherwise, $s$ is undefined.*

**Operation type `Write`:**

(c) *If $p$ executes a successful $M$.`chngCurBlock`$(x, y)$ at line* **255** *during $Op$ in step $i$ of $H$, then $s = (y, i, 0)$. (If $Op$ is pending in $H$, its completion returns $OK$.)*

(d) *Else if $Op$ is complete in $H$, and $p$ executes a failed $M$.`chngCurBlock`$(b, y)$ at line* **255** *during $Op$, and a successful $M$.`chngCurBlock`$(b, x)$ (by any process) occurs in step $i$ of $H$, then $s = (x, i, -p)$. (Block $x$ is well-defined by the specification of the block manager type.)*

(e) *Otherwise, $s$ is undefined.*

Next, we define $s_i = (x_i, t_i, ...)$, $Op_i$, $p_i$, and the candidate linearization $\bar{H}$ as in Section 6.1.1. (In particular, we continue to order timestamps according to Definition 6.9.)

**Lemma A.4.** *The analogs of Lemma 6.10 and Lemma 6.13 (properties (a) and (b) of linearizability – sequential completion and order preservation) for $I_{EW}$ hold.*

*Proof.* Both lemmas follow for $I_{EW}$ by proofs analogous to those given in Section 6.1. References to Lemmas 6.5 and 6.6 are replaced by references to Lemmas A.1 and A.2. References to Definition 6.8 are replaced by references to Definition A.3. The analogs of Definition 6.8 (d)–(e) are Definition A.3 (c)–(d). The analogs of lines **141** and **156** are lines **248** and **255**. □

**Lemma A.5.** *The analog of Lemma 6.15 (property (c) of linearizability—conformity to type $\tau_{ECAS\text{-}W}$) for $I_{EW}$ holds.*

*Proof.* Since conformity to a type is a safety property it suffices to consider finite $\bar{H}$. Let $k = |\bar{H}|$. Define $x_0$, $x_{k+1}$, $t_0$ and $t_{k+1}$ as in the proof of Lemma 6.15. We will show that for any $i \in \mathbb{N}$, $0 \leq i \leq k$:

(a) If the timestamp $t_i$ does not fall under Definition A.3 (d), then for any integer $t \in [t_i, t_{i+1})$, $x_i \triangleright B = \nu_i$ holds in state $H[t]$.

(b) If $i > 0$, then the response of $Op_i$ is the correct response for an operation execution of that type applied in state $\nu_{i-1}$.

Part (b) implies the lemma, but we require both parts for induction. Now let $S(i)$ denote parts (a)–(b). Note that in $H$, the current block and state of $B$ in that block are changed only by an execution of line **255**, line **261**, line **266**, or line **271**, which is an atomic step that defines the timestamp of an operation execution on the target object in $\bar{H}$. Therefore, the current block

and state of $B$ in that block do not change between atomic steps $t_i$ and $t_{i+1}$ in $H$. This, in turn, implies that to prove part (a) of $S(i)$, it suffices to prove that $x_i \triangleright B = \nu_i$ in state $H[s_i]$—and that is all we do in the inductive step that follows.

For $S(0)$, part (a) follows from our earlier definition of $x_0$ as the initial block and $t_0 = 0$, as well as the initialization of $x_0 \triangleright B$ to the initial state of type $I_{EW}$. Now suppose that $S(i-1)$ holds for some $i$, $0 < i \leq k$, and consider $S(i)$. We proceed by cases on how $s_i = (x_i, t_i, ...)$ was obtained, noting that $x_i = x_{i-1}$ except possibly when $s_i$ falls under Definition A.3 (c) or (d).

**Case A:** $Op_i$ falls under Definition A.3 (a). In this case, $s_i = (x_i, t_i, 0)$ for some $t_i$, $Op_i$ is a `Read` or `LL` or `ECAS` operation execution, and $p_i$ applies the corresponding operation to $x_i \triangleright B$ in step $t_i$.

For $S(i)$ (a), first note that $x_i = x_{i-1}$ and that $Op_{i-1}$ does not fall under Definition A.3 (d), otherwise by our construction of $\bar{H}$, $Op_i$ would be a `Write` operation execution falling under Definition A.3 (c) or (d). Thus, it follows from $S(i-1)$ (a) that $x_i \triangleright B = \nu_{i-1}$ in state $H[t_i - 1]$. Consequently, $p_i$'s operation in step $t_i$ changes the state of $x_i \triangleright B$ to $\nu_i$, as wanted.

For $S(i)$ (b), note that $Op_i$ returns the response of step $t_i$, where $p_i$ applies an operation execution of the same type as $Op_i$ to $x_i \triangleright B$. Since we showed that $x_i \triangleright B = \nu_{i-1}$ in state $H[t_i - 1]$, this response is correct for $Op_i$ in $\bar{H}$.

**Case B:** $Op_i$ falls under Definition A.3 (c). In this case, $s_i = (x_i, t_i, 0)$ for some $t_i$, $Op_i$ is `Write`, and $p_i$ executes a successful $M$.`chngCurBlock(...,` $x_i$`)` at line **255** in step $t_i$ of $H$.

$S(i)$ (a) follows by the action of step $t_i$ by $p_i$ in $H$, which makes $x_i$ the current block, and from the prior initialization of $x_i \triangleright B$ to $val_i$ (i.e., the argument of $Op_i$) at line **254**.

$S(i)$ (b) holds since $Op_i$ returns $OK$ at line **258**.

**Case C:** $Op_i$ falls under Definition A.3 (d).

Here $S(i)$ (a) holds trivially since $Op_i$ falls under Definition A.3 (d).

$S(i)$ (b) holds since $Op_i$ returns $OK$ at line **258**.

□

*Proof of Theorem 8.1.* The theorem asserts that the implementation $I_{EW}$ satisfies Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 6.4. Furthermore, each operation execution on the target object incurs $O(1)$ RMRs in the CC and DSM models.

Specification 6.1 under Condition 6.4 follows from Lemma A.4 and Lemma A.5. RMR complexity and Specification 6.2 follow from the same arguments as in the proof of Theorem 6.16. □

## A.6 Analysis from Section 8.1.1

**Lemma A.6.** *The analog of Lemmas 6.5 and 7.5 for $I'_{EW}$ holds.*

*Proof.* This follows by the same proof as given in Section 6.1. □

**Lemma A.7.** *The analog of Lemmas 6.6 and 7.6 for $I'_{EW}$ holds.*

*Proof.* This follows by the same proof as given in Section 6.1, with any reference to Lemma 6.5 replaced by a reference to Lemma A.6. □

**Lemma A.8.** *The analog of Lemma 7.7 holds for $I'_{EW}$.*

*Proof.* This follows by a proof analogous to the one given in Section 7.2 since $I'_{EW}$ uses the same implementations of subroutines `TryToReuseBlock`, `DoneReusingBlock`, `HelperBegin` and `HelperEnd` as $I_{EW}$, and since these functions are called in an analogous manner. Any reference to Lemma 7.5 in the proof is replaced by a reference to Lemma A.6. □

**Lemma A.9.** *For any history $H$ of implementation $I'_{EW}$, any process $p$, and any block $x$, if $p$ is at line* **277** *during a call to* `HelperCC(`$x, val$`)` *then:*

*(a) $p$ is the only process that has accessed $x \triangleright B$; and*
*(b) $x$ is current from the point when $p$ last called $M$.`getCurBlock()` until $p$ makes a call to* `DoneReusingBlock(`$x$`)` *at line* **280**.

*Proof.* Note that if $p$ is at line **277** during a call to `HelperCC(`$x, val$`)`, then by the test at line **275**, $p$ is the process that allocated $x$. Furthermore, $p$ has completed a call to `TryToReuseBlock(`$x$`)` that returned `true`, and $p$ has not subsequently made a call to `DoneReusingBlock(`$x$`)`. Consequently, no process $q \neq p$ has completed a call to `HelperBegin(`$x$`)` by Lemma A.8 (specifically the analog of Lemma 7.7 (a) for $I'_{EW}$). Since a process must complete a call to `HelperBegin(`$x$`)` before accessing $x \triangleright B$, this implies part (a).

It follows similarly that no process $q \neq p$ completes a call to `HelperBegin(`$x$`)` until $p$ makes a call to `DoneReusingBlock(`$x$`)` at line **280**. This implies part (b) because no process applies a successful $M$.`chngCurBlock(`$x, ...$`)` between $p$'s last call to $M$.`getCurBlock()`, which returns $x$, and $p$'s subsequent call to `DoneReusingBlock(`$x$`)` (if it occurs); $p$ itself does not do this by the algorithm, and no $q \neq p$ does so because by Lemma A.7 that would imply $q$ calls $M$.`chngCurBlock(`$x, ...$`)`, which can only happen after $q$ completes a call to `HelperBegin(`$x$`)`. □

**Lemma A.10.** *The analogs of Lemma 6.10 and Lemma 6.13 (properties (a) and (b) of linearizability – sequential completion and order preservation) for $I'_{EW}$ hold.*

*Proof.* Both lemmas follow for $I'_{EW}$ by the same proofs as in Section 6.1, with reference to Lemmas 6.5 and 6.6 replaced by references to Lemmas A.6 and A.7. □

To prove linearizability, we define for any history $H$ of $I'_{EW}$ a candidate linearization $\bar{H}$ as in Appendix A.5, except that we augment the definition of timestamps (Definition A.3). That is, we add a new clause (between clause (d) and clause (e)) for a `Write` operation execution $Op$ in $H$ where line **277** is reached:

(g) Else if $p$ re-initializes $x \triangleright B$ at line **277** for some block $x$ during $Op$ in step $i$ of $H$, then $s = (x, i, 0)$. (If $Op$ is pending in $H$, its completion returns $OK$.)

**Lemma A.11.** *The analog of Lemma A.5 (property (c) of linearizability—conformity to type $\tau_{ECAS\text{-}W}$) for $I'_{EW}$ holds.*

*Proof.* We modify the proof of Lemma A.5 by extending the case analysis as follows:

**Case D:** $Op_i$ falls under Definition A.3 (g). In this case, $Op$ is a `Write` operation execution where $p_i$ re-initializes $x_i \triangleright B$ in step $t_i$ at line **277**.

It follows from Lemma A.9 (b) that $x_i$ is current in state $H[t_i - 1]$. Consequently, $S(i)$ (a) follows from the action of $p_i$'s step at time $t_i$. This step ensures that $(x_i \triangleright B).V = val$ in state $H[t_i]$ by re-initializing the value of $x_i \triangleright B$. Similarly, it ensures that $(x_i \triangleright B).Linked[p_i] = \mathsf{false}$ in state $H[t_i]$. (Recall our prior explanation of the re-initialization operation at the end of Section 8.) As for the other elements of $(x_i \triangleright B).Linked[1..N]$, these are all $\mathsf{false}$ in states

$H[t_i - 1]$ and $H[t_i]$ by the initialization of $x_i \triangleright B$, by Lemma A.9 (a), and since the re-initialization operation execution does not write them.

$S(i)$ (b) holds because $Op_i$ returns $OK$.

□

**Lemma A.12.** *The analog of Lemma 7.12 (termination) for $I'_{EW}$ holds*

*Proof.* This follows by a proof analogous to the one given in Section 7.2. References to Lemma 7.7 are replaced by references to Lemma A.8. □

**Lemma A.13.** *The analog of Lemma 7.13 ($O(1)$ RMR cost for a process to access the block manager and allocator) for $I'_{EW}$ holds.*

*Proof.* This follows by a proof analogous to the one given in Section 7.2. References to Lemma 7.5 are replaced by references to Lemma A.6. □

**Lemma A.14.** *For any history $H$ of $I'_{EW}$, for any block $x$ accessed in $H$, and for any process $p$, the number of RMRs that $p$ incurs while accessing block $x$ in $H$, not including the field $B$, is:*

- *$O(1)$ in the CC model with write-back caching; and*
- *$O(1 + m)$ in the CC model with write-through caching, where $m$ is the number of write-like operation executions in $H$ on the target object during which $p$ accesses block $x$.*

*Proof.* This lemma is the counterpart of Lemma 7.14 for $I'_{EW}$, but not its analog, because we ignore RMRs incurred while accessing $B$, which is the counterpart of $V$ in Section 7.2. (The analog does not hold because a process may incur arbitrarily many RMRs in a history where only block $x$ is accessed.) Still, the lemma follows by an analogous proof, where we drop the case that considers the block field $V$. References to Lemma 7.7 are replaced with references to Lemma A.8. □

**Lemma A.15.** *Implementation $I'_{EW}$ satisfies the locality property in the write-through and write-back CC model (see Definitions 7.2 and 7.3).*

*Proof.* Consider any history $H$ of $I'_{EW}$, and consider the linearization $\bar{H}$ of $H|O_\tau$ defined in our proof linearizability (see Lemma A.11), where $O_\tau$ is the target object. To prove the locality property, we will show that $p$ incurs $O(1)$ RMRs in $H$ while executing the counterparts of certain operation executions in $\bar{H}$, as in in the proof of Lemma 7.15. (As before, the case when $p$ does not access any block during some operation execution is discharged easily.) To that end, we will break down the analysis into two parts: accesses to the base object $B$ in blocks, and accesses to all other shared objects.

**Property (R)** (Definition 7.2). We must consider the write-through and write-back CC models. Fix process $p$ and a sequence $\bar{H}'$ of consecutive read-like operation executions in $\bar{H}$. Let $H'$ denote the sequence of atomic steps (which access base objects) in $H$ corresponding to $\bar{H}'$.

First, we will show that $p$ accesses at most one block, say $x$, in $H'$. Suppose otherwise. By our definition of $\bar{H}'$ and $H'$, $p$ does not call $M$.`chngCurBlock` or `AllocBlock()` in $H'$, since that can only occur during a `Write` operation execution. Furthermore, calls to $M$.`getCurBlock()` by $p$ in $H'$ return at most one block, otherwise between two such calls by $p$ there is a successful $M$.`chngCurBlock` in $H$, and so there is a `Write` operation execution in $\bar{H}$ that is linearized between $p$'s first and last operation execution in $\bar{H}'$, contradicting the assumption that $\bar{H}'$ contains read-like operation executions only. Thus, by Lemma A.13 and Lemma A.14, $p$ incurs $O(1)$ RMRs accessing shared objects other than $B$ in $H'$.

It remains to consider $B$. Note that between any two base object atomic steps on $x \triangleright B$ by $p$ in $H'$, there is no write-like operation on $x \triangleright B$ by any other process, otherwise again we reach a contradiction since there is a write-like `ECAS` operation execution in $\bar{H}$ that is linearized between $p$'s first and last operation execution in $\bar{H}'$. Thus, by locality property (R) of $x \triangleright B$, $p$ incurs $O(1)$ RMRs accessing $x \triangleright B$ in $H'$, as wanted.

**Property (W)** (Definition 7.3). We need only consider the write-back CC model. Fix process $p$ and a sequence $\bar{H}'$ of consecutive operation executions by $p$ in $\bar{H}$. Again, let $H'$ denote the sequence of atomic steps in $H$ corresponding to $\bar{H}'$.

First, we will show that $p$ tries to change the current block at most once in $H'$. Suppose, for contradiction, that $p$ does this in the counterparts of operation executions $Op$ and $Op'$ in $\bar{H}'$. Arguing as in the proof of Lemma 7.15, property (W) (with references to Lemma 7.7 replaced by references to Lemma A.8), this implies that there is an operation execution in $\bar{H}'$ between $Op$ and $Op'$ by a process different from $p$, which contradicts the definition of $\bar{H}'$. Next, note that calls to $M$.`getCurBlock` by $p$ in $H'$ return at most two distinct values. This is because if three values are returned, then there are at least two successful calls to $M$.`chngCurBlock` in $H$ between the first and last step in $H'$, where at most one is by $p$ (as argued above), and the other (by a process different from $p$) coincides with the timestamp of a `Write` that appears between the first and last operation execution in $\bar{H}'$, which contradicts $\bar{H}'$ containing operation executions by $p$ only. Thus, $p$ accesses at most three blocks in $H'$, and so by Lemma A.13 and Lemma A.14, $p$ incurs $O(1)$ RMRs accessing shared objects other than $B$ in $H'$.

It remains to consider $B$. Note that for any block $x$ process $p$ accesses, and between any two base object atomic steps on $x \triangleright B$ by $p$ in $H'$, there is no atomic step at all on $x \triangleright B$ by any other process. To see this, note that otherwise by our construction of $\bar{H}$ there would be an `ECAS`, `LL`, or `Read` operation execution in $\bar{H}$ by a process different from $p$ that is linearized between $p$'s first and last operation execution in $\bar{H}'$, which contradicts $\bar{H}'$ containing operation executions by $p$ only. Thus, by locality property (W) of $x \triangleright B$, $p$ incurs $O(1)$ RMRs accessing $x \triangleright B$ in $H'$, as wanted. □

*Proof of Theorem 4.5.* The theorem asserts that the implementation $I'_{EW}$ satisfies Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 6.4. Furthermore, each operation execution on the target object incurs $O(1)$ RMRs in the CC model. Finally, $I'_{EW}$ satisfies the locality property in the write-through and write-back CC models (Definitions 7.2 and 7.3).

Specification 6.1 (linearizability) under Condition 6.4 follows directly from Lemma A.10 and Lemma A.11. Specification 6.2 (termination) under Condition 6.4 follows from Lemma A.12. $O(1)$ RMR complexity follows from Lemma A.13, Lemma A.14, as well as the fact that during any operation execution on the target object, a process accesses at most two blocks and accesses the field $B$ at most once per block per operation execution. Locality follows from Lemma A.15. □